Sassenrath Research

Guru's Guide

To The Commodore Amiga

Meditation #1 – Interrupts

Guru's Guide

To The Commodore Amiga

Meditation #1 – Interrupts

Carl Sassenrath

Special thanks to Cynthia L. Sassenrath for her help and support.

A Product of Sassenrath Research,
P.O. Box 1510, Ukiah, CA 95482 USA

Copyright © Sassenrath Research, 1988.

All Rights Reserved.

This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without the prior written consent from Sassenrath Research.

DISCLAIMER

THIS DOCUMENT IS SOLD WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

SASSENRATH RESEARCH DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF, OR THE RESULTS OF THE USE OF, THIS DOCUMENT IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, CURRENTNESS, OR OTHERWISE. THIS DOCUMENT IS PROVIDED "AS IS" AND THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY IS ASSUMED BY YOU. IN NO EVENT WILL SASSENRATH RESEARCH OR CARL SASSENRATH BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

SOME LAWS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

Information in this document is subject to change without notice and does not represent a commitment on the part of Sassenrath Research.

Guru's Guide is a trademark of Sassenrath Research
Amiga is a trademark of Commodore-Amiga, Inc.
Aztec C is a trademark of Manx Software Systems

PRINTED IN THE U.S.A.

Table of Contents

1 – Introduction

1.1 The Cave 1
Early Documentation 2
ROM Kernel Manual 3
Responsibility 4
1.2 The Trail 5
The Author 5
The Reader 6
Examples 6
Your Support 7
1.3 Overview 8
Design Goals 9
Flow of Control 10
1.4 Applications 12
Devices 12
Libraries 12
Programs 12

2 – Principles

2.1 Hardware 13
Sources 14
Controller 16
Hardware Priorities 20
Master Enable 20
Direct IPL and NMI 21
8520 CIAs 21
2.2 CPU 23
68000 CPU 23
Auto-Vectors 25
Supervisor Mode 25
Supervisor Stack 27
CPU Priority Level 28
Nested Interrupts 29
2.3 Exec 31
Spurious Interrupt Rejection 31
Decoding 31
Dispatching 32
Handling 32
Sharing 33
Task Dispatching 35

Pseudo-Priorities 36
2.4 Software Interrupts **38**
Environment 38
Priorities 39
Requesting 39
2.5 Task Interrupts (Exceptions) **40**
Bad News 40

3 – Programming

3.1 Languages **43**
C Language 45
Assembly Language 46
Combination 46
3.2 Data Structure **47**
List Node 48
Data Pointer 49
Code Pointer 49
Initializing 49
Clean-up 51
3.3 System Functions **52**
SetIntVector 52
AddIntServer 55
RemIntServer 57
Cause 59
3.4 Monitor **61**
3.5 Servers **64**
Environment 64
Chaining 65
Server Priorities 66
Examples 67
3.6 Task Signals **70**
Functions 71
Example 71
Coprocessor Example 73
3.7 Controller **77**
Include Files 77
Set/Clear 78
Support Functions 80
Master Enable 80
3.8 Handlers **82**
Setup 82
Clean-Up 83

Environment *84*
Audio Channel Example *84*
Serial Port Example *87*
3.9 CIA Interrupts ***91***
Functions *92*
AddICRVector *93*
RemICRVector *94*
AbleICR *95*
SetICR *95*
ICR Bits *96*
CIA Example *96*
3.10 Software Interrupts ***99***
Priorities *99*
SWI Example *99*
Message Example *100*

1 - Introduction

1.1 The Cave

It's one of those mind impressions, the autumn of 1983. A dim, windowless burrow aglow with phosphorescent portals. Shadows cast adrift among shining eyes. Echos of dancing keycaps accompanying a choir of cooling fans. A shout: "I'm in the queue for the machine" or "Out of my way, I'm a busy guy!" A faded cartoon taped to a heavy bookshelf warns not to feed the programmers. The aura of inventing the future: the software cave at Amiga Computer, Inc.

In a corner, huddled around a narrow stream of light, Jay, Glenn, Dave, Dave, and Ron coax the intricate "friend" into operation. The custom chips were but circular towers, 2500 little CMOS meatballs atop a large portion of silver-blue spaghetti. Truly a masterpiece. To any observer the Amiga was clearly to be a mainframe.

We were all dreamers. No one knew what the Amiga would become, but those were the days when vision ruled our souls: Bob, Dale, RJ, Sam, Kodiak, Rob, Sheryl, Neil, and I. We believed in the 500 dollar high-tech dual bus 25 DMA channel graphics and sound magic machine. To us it would be fitting for the quick little data mover to inhabit that empty space between the flickering TV tube and the darting VU meters. The country... no, the world, must view and hear this bouncing ball.

And so from scratch, with few chisels to carve our granite, we began to shape the Amiga kernel and software libraries. Like the hardware on which it ran, the software would be a massive undertaking. As the system grew, the resources of the employee owned Amiga Computer shrank, and our hopes for real tools faded. Finally in the fall of 1985 – with the aid of Commodore

Business Machines, 30 Sun's, three C compilers, and five assemblers – Kickstart version 1.0 was "ready".

The intriguing Amiga and its new software system attracted the attention of engineers and artists the world over. Unfortunately, version 1.0 frequented the guru a little too often. It was not until 1.1 that the system became a worthy product. Later, version 1.2 offered valuable enhancements and fixes. By then the software system included more than 256 KBytes of ROM resident code. A consistent architecture of libraries and devices united more than a dozen modules within the Kickstart memory of the Amiga.

Early Documentation

The Amiga is a system born of the grand technological promises of Very Large Scale Integration (VLSI). During development the hardware and software complexities of the system quickly outgrew our capacity to remember each detail. This situation further deteriorated as third-party developers were urged to contribute. Some form of written reference manual had to be produced if serious progress was to continue.

From the early days, whenever the hardware design advanced, the documentation group was usually not far behind. Early versions of the hardware reference manual relieved much of the demand for written information, and they were widely distributed among developers.

Unfortunately, the software design was in a completely different state. In those days the software consisted mostly of abstract ideas rotating through thick gray matter. The design was certainly not stable enough to be placed into a written form. So, for the first six months software progress went undocumented.

Of course, the Amiga kernel was a structured system built from nested layers of modules. Higher level modules were expected to take full advantage of lower level functions. To make this possible, a minimal amount of documentation was required. This drove us to create a technique for source code embedded documentation.

Whenever a new kernel function was created, a comment in a standard form was placed immediately before the function in the source code. For a module designer, these comments were convenient to maintain – separate documentation was not required. Every few weeks a special program would scan all source files and produce a document by extracting the function

comments. Because documents were generated in this "automatic" fashion, the result was called an AutoDoc[1].

The software documentation began to get more serious attention when it came time to specify the design for AmigaDOS. The plan was to farm out the design of DOS to a contract system house. A thirty-five page specification for CAOS (Commodore Amiga Operating System) was created[2]. The system house was also to be responsible for the design of the Amiga's graphical user interface. Another document was created in an attempt to specify the form of the user interface.

ROM Kernel Manual

By the summer of 1984, the software kernel design was starting to stablize. The general architecture of the system had been well established and several modules were actually functioning. The documentation team began to focus their efforts on the software side of the Amiga and the *ROM Kernel Manual* came into being.

By the first system release the basic descriptions of functions and structures had accumulated into more than two thousand pages of technical prose. This massive collection was finally published by Addison Wesley as the *Amiga ROM Kernel Reference Manual* and other works.

The ROM Kernel Reference Manual had to cover a wide territory. It was only practical that some topics would need to be discussed in more detail than others. Typical software applications required information about the user interface, DOS, graphics, and printer support. Unfortunately as a result, the more obscure low-level kernel functions did not get much attention, and they were not described in adequate detail.

A few of us hoped that within a couple years, some talented, insightful engineer/writer would fill the voids and round out the material available to serious programmers. Now, more than three years after the Amiga introduction, only a few minor attempts have been made. Technical books have been published, but none of these focused on the architecture and principles of

---

[1] AutoDocs are still used as the "official" description for all Amiga system functions. They are made available to software developers as part of each system release.

[2] This system was never actually produced. AmigaDOS today is substantially different from this early specification.

the system. These books were pragmatic in nature, discussing library functions, describing parameters, and giving simple examples. To venture beyond, you had to cut your own path and disassembly, experimentation, hacking, and blind guessing became widespread.

Responsibility

Programming the Amiga is much like programming any other microcomputer. You design a program to interact with the user and produce some net effect. Of course, at the same time your program must comply with the set of rules or constraints established by the operating system. It is your responsibility as a programmer to abide by these rules. If you decide to ignore them, chances are good that your program will eventually fail or the computer will crash.

When using a multitasking operating system like the Amiga's, a number of additional programmer responsibilities are imposed. The Amiga kernel operates in an open, unprotected memory configuration. This means that in addition to problems your program may create for itself, it has the capability of disrupting any or all of the other programs currently running. One defective program can bring the whole system down.

In this type of environment, it is not a good practice to guess. Correct system usage is dictated by theory, not from mere experimentation. A program built experimentally may at first seem to function correctly, but if it violates even the smallest rule, it may place the integrity of the entire system at risk.

With this in mind, it becomes clear that the ROM Kernel Manual is not enough for a serious programmer. It doesn't describe the necessary rules, constraints, and techniques for low-level system programming. It leaves far too much to the imagination of the programmer.

1.2 The Trail

This volume of the *Guru's Guide* is devoted exclusively to the topic of interrupts on the Amiga. It attempts to fill the void left by chapter five of the original *ROM Kernel Reference Manual: Exec*. It supplies a much greater level of detail for all aspects of interrupt processing.

The primary objective of this volume is to provide a thorough description of interrupts from the perspective of a software designer. Additionally, it strives to help programmers create bug-free interrupt control code with a minimal amount of effort.

The remainder of this guide is divided into two parts:

Chapter two, *Principles*, acquaints you with the theoretical side of interrupts. It attempts to give you a general view of the system without getting bogged down in punctilious programming detail. It discusses both the hardware and software elements in great depth.

Chapter three, *Programming*, applies the principles described in the previous chapter toward the goal of creating working code. Here the focus is on the data structures, system functions, and programming techniques needed to process interrupts. Many working examples are included to serve as a starting point for further development.

The Author

In 1983 Carl Sassenrath left Hewlett Packard to join Amiga Computer, Inc. as the Manager of Operating Systems. At HP he was involved in the design of operating systems, graphical user interfaces, and object-oriented languages. During his years there he concluded that it was possible to develop a small, quick multitasking operating system that would be a balance of power and simplicity.

At Amiga, Carl was the principle designer and implementer of the Multitasking Executive (Exec). As such, he was responsible for the low-level software architecture of the Amiga.

In 1985 after losing the battle to prevent the adoption of the current AmigaDOS, Carl resigned from his position as Manager of Operating Systems. He did not wish to become associated with AmigaDOS in any way. Instead, he became Manager of

Advanced Execution Environments, where he pushed for the use of object-oriented operating systems and multi-processor architectures.

Frustrated, Carl left Amiga in late 1985 to develop his ideas on object-oriented systems and languages. In the fall of 1986 he joined Apple's Advanced Technology Group with the hope of creating future operating system technologies.

In early 1988 Carl retired from the Silicon Valley lifestyle and returned home to the peaceful redwood forests of Northern California. He now heads a small company dedicated to the development of high technology software.

The Reader

There are several approaches to climbing a mountain; the route you pick will depend on your skill level and what you want to get out of the hike. This guide was written primarily for experienced software designers and system programmers who require a comprehensive treatment of the Amiga interrupt system.

To completely understand this guide, you must have a good working knowledge of the Amiga software architecture. In addition, you will need to know the basics of systems programming on the Amiga. You will also find familiarity with C and Assembly code to be helpful.

Although this guide was not created for the novice or casual programmer, it does not discourage an educational approach to the material. Any enthusiastic soul with enough energy will benefit from a serious study of this text. We recommend that you learn about the operating system from other supplemental books before attempting to enhance the program examples provided in this guide.

Examples

The *Programming* chapter includes many examples of working interrupt code. An attempt was made to keep each of these examples as short and to the point as possible.

All of the examples have been compiled and executed on an Amiga. They were compiled with Manx, Aztec C 3.6A[3] using include files from Kickstart release 1.2. They were linked with the Manx 32 bit C library, C32. The 32 bit integer size is used as the "int" default, so if you switch to 16 bit integers, the examples will need to be altered.

Some of the examples contain sections written in 68000 assembly code. These sections take advantage of the Aztec C "#asm" preprocessor directive. Other compiler brands may use a different name for this option. Consult your compiler's user manual. If you need to, you can assemble these sections separately and link them to the final program file.

Your Support

If you have suggestions for improvements to this guide, or if you've found errors or bugs, we would be very interested to hear about them. If there are other topics that you would like to see covered in future guides, let us know about those as well. Our mailing address is printed on the back side of the title page.

Of course, we cannot make any promises or guarantees. The future of the *Guru's Guide* will depend entirely on the success of this first meditation. You can help make it a success by telling others how much you like it and by encouraging them to purchase it.

[3]If you are using an earlier release of the Aztec compiler, we strongly urge you to update it.

1.3 Overview

The Amiga Multitasking Executive (Exec) is responsible for the basic run-time environment of software on the Amiga. Exec establishes the fundamental system architecture and provides the low-level services needed to support the ROM kernel. It consists of modules for system startup, tasks, devices, libraries, messages, interrupts, resources, memory, traps, alerts, and lists.

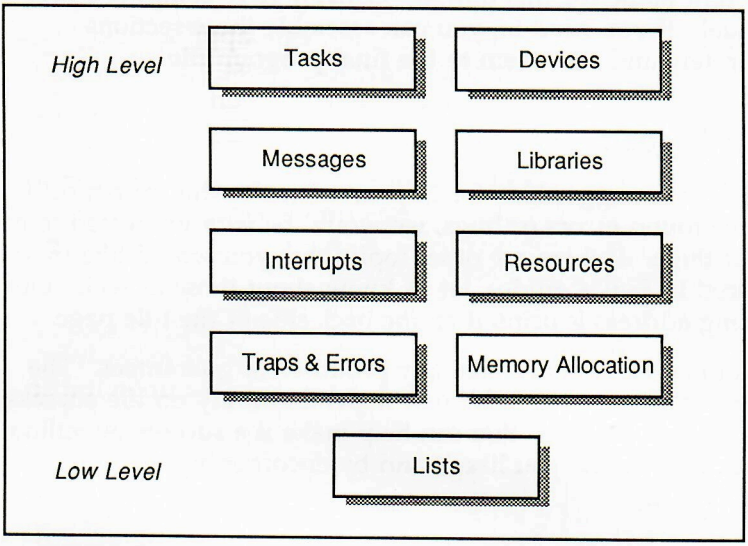

Figure 2.1 – Modules of Exec

Handling interrupts is one of the primary duties of Exec. Exec provides all of the organizing, structuring, prioritizing, decoding, and dispatching to make interrupts functional.

The interrupt system can be thought of as the core for event driven activities in Exec. To a large degree interrupts drive the entire software event system. This is especially important to operation on the Amiga due to its many DMA channels and multitasking.

Exec will be key to your use of interrupts on the Amiga. It defines the data structures, supplies the relevant functions, and performs the internal operations necessary for interrupt processing.

Design Goals

As one of the primary sub-systems of the Amiga kernel, the Exec interrupt system was designed to meet a number of general goals. It was meant to be:

Consistent with the philosophy and structure of the rest of Exec. From the beginning it was planned to use standard Exec list nodes with names and priorities.

Fast enough for most software applications, including video games. Fast interrupts also increase the overall system efficiency by reducing overhead.

Simple to use and understand. Keep it small and to the point. It uses simple structures and functions to make interrupt programming easy.

Flexible enough to expand with the needs of the machine and changes in hardware. New layers of software could be built upon it if so desired.

A number of other specific technical goals also seemed important during design. The system should support interrupt priorities, the sharing of interrupts, nested interrupts with higher priorities, and software generated interrupts.

There were also features that we did not desire or could not afford. The interrupt system was not intended to be resistant to abuse – Exec would not be forgiving when used improperly. It would not conform to any standard architecture of the time. It was not to be a big system; there would be no resource management, user identifiers, memory protection, or security levels. The system would not clean-up after tasks that failed.

Flow of Control

As a brief introduction to the interrupt system, consider the flow of control initiated by an interrupt. Figure 1.2 illustrates what happens when an interrupt occurs.

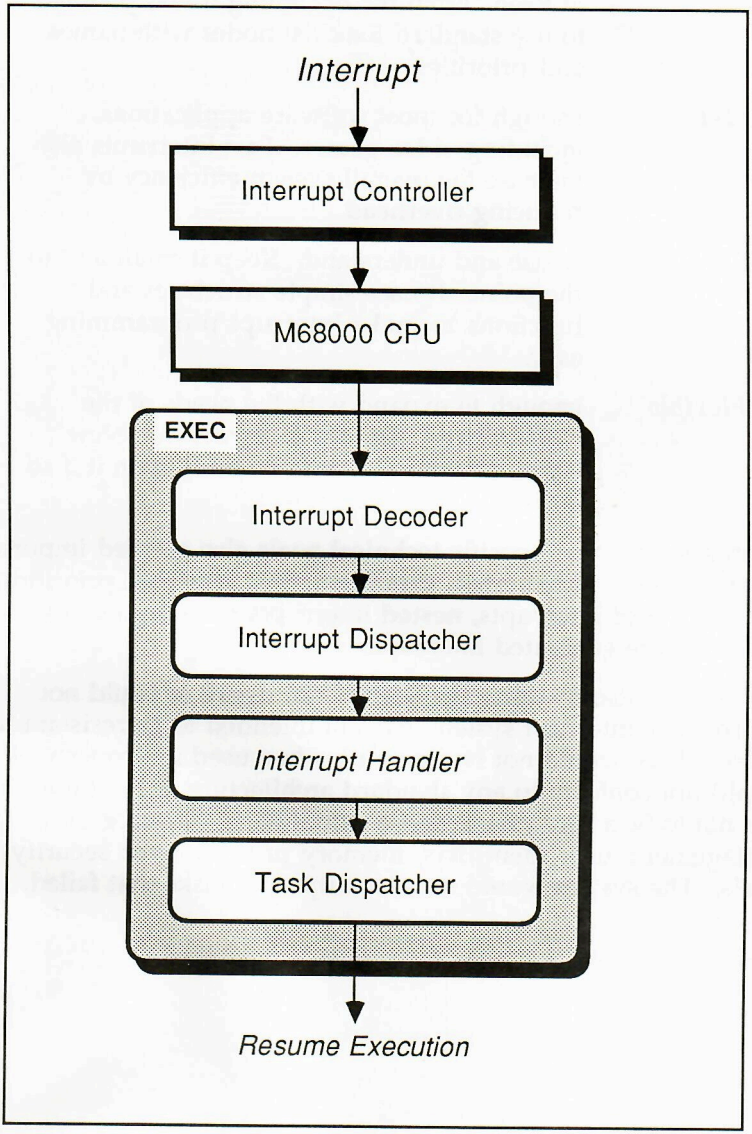

Fig 1.2 – General Flow of Control

The flow of control can be broken down into seven actions:

1. An interrupt occurs. It may be generated by a hardware device pulling on an interrupt line or by software writing to the interrupt controller.
2. The interrupt controller prioritizes and flags the interrupt. It classifies the interrupt into one of six levels.
3. The M68000 CPU is informed of the interrupt. The code currently executing is temporarily halted, its state is saved, the CPU enters supervisor mode, and control is passed to Exec.
4. Exec decodes the interrupt by examining the interrupt controller.
5. Exec dispatches the appropriate interrupt handler. This handler performs whatever operations are required to process the interrupt. (If the interrupt is of the shared type, an Exec handler will further dispatch interrupt processing code.)
6. When the handler has finished, it returns control to Exec. Exec then determines whether it should resume the previously interrupted code or dispatch a different task.
7. Execution is resumed. If there is nothing to execute the CPU is stopped until the next interrupt.

This guide will discuss each of these actions in great detail.

1.4 Applications

Devices

Device drivers are the most common place for interrupt processing code. Most devices need to synchronize various types of hardware events with software. For example, the Amiga floppy disk hardware will generate an interrupt each time the transfer of data to or from the disk has been completed.

Exec specifies a common form for all device drivers. This specification includes a common I/O structure, interface functions, and device commands. Because Exec devices are basically asynchronous in nature, they are well suited to interrupt processing. Programs interface to devices at a high level, and they are totally unaware of any interrupts that may take place within the device itself.

Libraries

Exec is designed to support a special memory resident form of shared libraries. Although it is more rare, interrupt processing will sometimes be performed within a library. For instance, the graphics library uses vertical blanking video interrupts to synchronize certain types of operations.

Programs

Although most programs do not normally require the direct use of interrupts[4], a program may occasionally need explicit control of an interrupt. It might require close control over the serial port or need a short software interrupt to handle an event.

Using interrupts within a program is also a common method of testing hardware or proving a software concept without the difficulties involved in writing a device driver. In this guide all of the examples will be of this variety.

[4]However, many programs use interrupts indirectly through devices or libraries.

2 - Principles

The Amiga is a busy machine. With its custom VLSI chip-set it has the capability of performing many operations at once. In addition to the activities of its Motorola 68000 microprocessor, the Amiga may simultaneously be displaying graphics images on the screen, playing music or sound effects on four separate audio channels, moving or manipulating blocks of data with the blitter, transferring data to or from floppy disk, sending or receiving data on its serial and parallel ports, synchronizing and controlling hardware with its coprocessor, receiving characters from its keyboard and mouse, and timing various intervals.

Each of these activities occurs on demand, at its own speed, without any knowledge of the other activities that share the same moment in time. Yet, all of these activities must at some point bid for the attention of their controlling software executing on the 68000 microprocessor. They achieve this through the use of interrupts. Only with a robust interrupt system does this degree of parallelism become practical.

This chapter describes the principles of the Amiga interrupt system in great detail. It covers both the hardware and software aspects of interrupts and defers the picky programming details to the next chapter.

2.1 Hardware

On the Amiga, most of the active hardware devices generate some form of interrupt. There are interrupts for video, audio, disk, blitter, coprocessor, keyboard, serial port, parallel port, timers, and external devices. All of these must pass through special interrupt control circuitry in order to capture the attention of the Motorola 68000 microprocessor.

To reduce development and production costs, the Amiga was designed to require a minimal amount of interrupt control circuitry. The primary hardware component for interrupt processing was built into the silicon of the *Paula* custom chip. It provides the basic control needed for enabling, requesting, servicing, and clearing interrupts. In addition, two *8520 Complex Interface Adaptors* (CIAs) perform a limited amount of interrupt processing. The Amiga also provides circuitry to support certain types of external interrupts generated on the Amiga expansion bus.

Figure 2.1 roughly illustrates the hardware system with all of its major components. Notice that Paula is the center for interrupt activities. The CIA devices feed into its external inputs, the Agnus custom chip communicates through a special bus, and Paula sends itself interrupts internally.

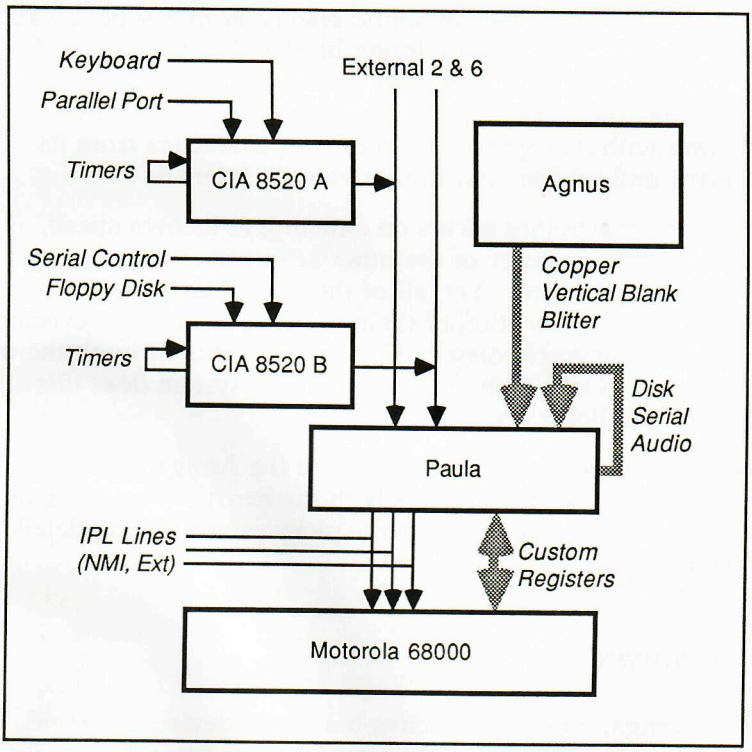

Fig 2.1 – Amiga Interrupt Hardware

Sources

On the Amiga, the custom chips (Agnus, Paula, Denise) account for much of the interrupt activity that occurs. These custom chips were designed to permit a significant amount of activity in parallel to the 68000 CPU. They contain more than 20 direct memory access (DMA) channels that operate independent of normal CPU operations. When they want to communicate their status (indicate that an activity has completed or that a special event has occurred), they do so by generating the appropriate interrupt:

Serial Port The Paula RS232 serial port can generate interrupts on two conditions. When the input buffer has received a new character, the *Read Buffer Full* (RBF) interrupt occurs. When the output buffer is ready for the next character to transmit, the *Transmitter Buffer Empty* (TBE) interrupt occurs.

Video Each time the video electron beam reaches the bottom of the screen, video is turned off while the beam returns to the top of the screen. This period is important to software for video timing and display updating. The *Vertical Blank* interrupt marks the beginning of this period.

Copper Occasionally it is necessary to know when the video beam reaches an arbitrary position on the screen. The Amiga coprocessor (Copper) has the ability to cause such an interrupt.

Blitter The Amiga blitter is basically a DMA data manipulator. When it has fulfilled a request, it can notify software with an interrupt.

Audio Sounds are generated on the Amiga by sending a block of audio data (DMA) to a digital to analog converter. Each of the four audio channels can cause an interrupt to indicate that it is ready for a new block of data.

Disk The floppy disk controller can generate two types of interrupts. It can flag that a DMA transfer of data to or from disk has finished. It can also cause an interrupt when a particular bit pattern was just read from the disk.

Of course the custom chips are not the only source of interrupts. The two 8520 CIAs also generate a number of interrupts. These CIAs provide the Amiga with timers and external interfaces (parallel and serial ports) – both of which can cause interrupts.

Table 2.1 summarizes the sources of interrupts on the Amiga. The names used here are the same as those used throughout the hardware and software documentation[1].

Name	Description
SOFTINT	software interrupt
DSKBLK	disk block done
TBE	transmitter buffer empty
PORTS	I/O ports & external (low)
COPER	graphics co-processor
VERTB	vertical blanking
BLIT	blitter done
AUD2	audio channel 2
AUD0	audio channel 0
AUD3	audio channel 3
AUD1	audio channel 1
RBF	read buffer full
DSKSYNC	disk sync pattern found
EXTER	external (high)
INTEN	master enable (see text)
NMI	non-maskable interrupt

Table 2.1 – Interrupt Sources

Two lines are provided for *external* interrupts. When an interrupt occurs within a CIA device, it is forwarded to Paula as an external interrupt. Paula handles these interrupts in a similar fashion to those which occur in the custom chips.

An external interrupt may also originate from hardware devices operating on the Amiga Expansion Bus. These interrupts occur on the same lines as the CIA interrupts, and it is left up to software to distinguish between them.

---

[1] Copper, COPER, and Coprocessor all refer to the same hardware device.

Controller

The Paula custom chip provides most of the non-video features found on the Amiga. It includes four stereo audio channels, a floppy disk controller, a serial UART port, two general purpose I/O ports, and an interrupt controller.

Fourteen interrupt sources feed into the *Amiga Interrupt Controller* (AIC) from custom chips and external devices. The interrupt controller is responsible for enabling, prioritizing, requesting, encoding, and clearing all of these interrupts. Interrupts are not generally fed to the M68000 CPU directly[2]. They usually pass through the AIC first.

The Exec system software, its libraries and devices use this controller extensively. The software controls and monitors the AIC through a set of memory mapped registers. Four of these registers are used to communicate with the AIC. Two registers are used for reading, two are used for writing[3].

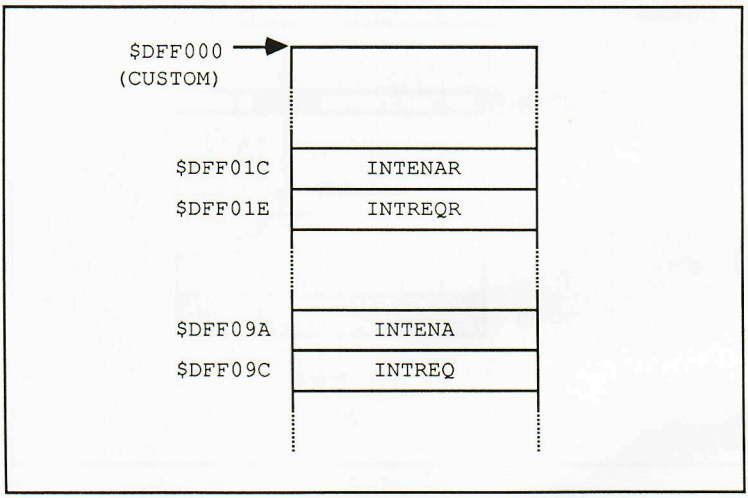

Fig 2.2 – Custom Chip Registers

The AIC registers are the interrupt enable registers INTENA (write) and INTENAR (read), and the interrupt request registers INTREQ (write) and INTREQR (read).

---

[2] With the exception of the non-maskable and direct interrupts.

[3] If these had been read/write registers, we could have gotten by with only two, but the extra hardware required would not justify its cost (so I have been told).

These registers always appear in the same absolute location within the Amiga memory address space. However, they are always referred to by name rather than by location.

Each register consists of 16 bits. The lower fourteen bits of each register are identical – each represents an interrupt source[4]. By changing these bits, each interrupt can be separately enabled, disabled, requested, and cleared.

When an interrupt is requested, Paula sets the corresponding bit in INTREQ. The INTREQ register is then masked with the INTENA register[5]. The logical AND of these two registers indicates which interrupts require processing.

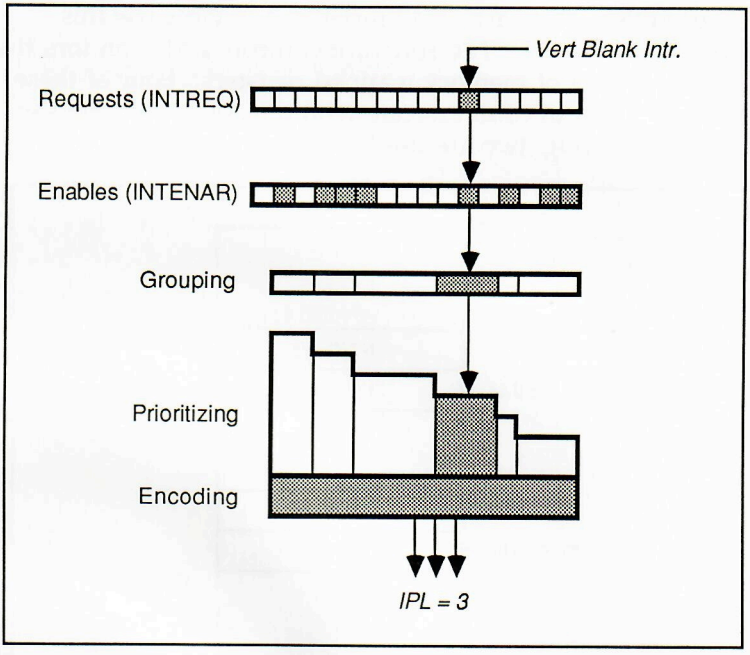

Fig 2.3 – Interrupt Masking, Grouping, Prioritizing

The AIC must now capture the attention of the CPU. It initiates a request to the CPU by encoding the priority level of the interrupt onto three lines, the 68000 *Interrupt Processing Level* (IPL) pins. Since 14 interrupts cannot be encoded onto the three CPU IPL

[4] Some of these bits are shared by a number of external devices.

[5] Exec addresses INTENA often. As you will see later, this is a higher performance means of controlling interrupts than using the CPU's supervisor mode.

lines (it would require four lines), the AIC collects the interrupts into priority groups.

There are seven priority groups. They are numbered one through seven, with the larger numbers indicating higher priority. Each group has one or more interrupts associated with it. When the AIC alerts the CPU, it encodes only the group number. The actual AIC interrupt must be determined later by the Exec interrupt decoding software.

Table 2.2 shows how interrupts are grouped by the AIC. The first column indicates the group number. The AIC bit number is shown in the second column.

Group	B	Name	Description
1	0	TBE	transmitter buffer empty
	1	DSKBLK	disk block done
	2	SOFTINT	software interrupt
2	3	PORTS	I/O ports & external (low)
3	4	COPER	graphics co-processor
	5	VERTB	vertical blanking
	6	BLIT	blitter done
4	7	AUD0	audio channel 0
	8	AUD1	audio channel 1
	9	AUD2	audio channel 2
	10	AUD3	audio channel 3
5	11	RBF	read buffer full
	12	DSKSYNC	disk sync pattern found
6	13	EXTER	external (high)
	14	INTEN	master enable (see text)
7	15	NMI	non-maskable interrupt

Table 2.2 – Interrupt Groups (Priorities)

Hardware Priorities

Occasionally, there will be more than one interrupt requested (and enabled) at the same time. The AIC will encode on the IPL lines the highest priority group that is currently pending. Any lower priority interrupts will be deferred.

The highest priority interrupt will remain asserted on the CPU IPL lines until the corresponding INTREQ or INTENA bit is cleared by the software. One of these two is usually done within the interrupt processing routine to reset the interrupt or disable it completely. Once a higher priority interrupt has been reset (or disabled), the AIC will set the IPL lines to the priority of the next highest requested and enabled interrupt.

If while processing an interrupt, another interrupt of a higher priority occurs, the IPL lines will change to the higher priority. The CPU will recognize this and a new interrupt will be generated. (See the section on nested interrupts.)

The system software further imposes a form of pseudo-priorities on these basic priorities. This will be discussed later.

Master Enable

Bit 14 of the AIC's enable register (INTENA) is designated as the *master* enable. This master overrides all other INTENA bit settings, but does not alter their values.

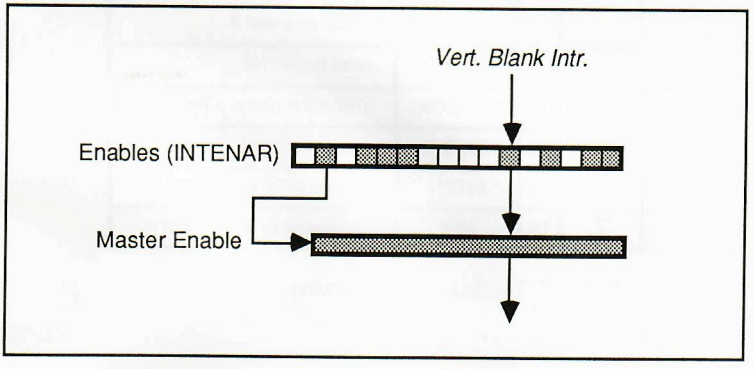

Fig 2.4 – Master Enable

When the master bit is set to zero, interrupts will not be generated from the AIC. As it turns out, this feature is very convenient for Exec. The software system frequently needs to disable all interrupts during critical sections when shared data

structures are being accessed. The master bit provides an quick and easy way to do this. The contents of the INTENA register do not need to be read and saved prior to disabling all interrupts. This is much more efficient than other schemes[6] and reduces the total overhead of the system considerably. Exec does not need to enter supervisor mode, nor does it need to save the state of each INTENA bit. The **Disable** and **Enable** Exec functions (and macros) make use of this extensively.

Direct IPL and NMI

As we have described, the AIC generates the necessary 68000 IPL signals needed to get the CPU's attention. It is possible for external hardware to tug on these IPL lines directly, and the CPU will recognize such an interrupt. This is does not normally occur on a stock Amiga. For example, externally hitting all three IPL lines will generate a 68000 non-maskable interrupt (NMI). This is the only way to generate NMI. The AIC cannot generate an NMI itself.

When the IPL lines are used directly, the AIC, its enabling, disabling, requesting, and prioritizing mechanisms are bypassed. This creates a number of serious problems for most of the system software, which relies entirely on the AIC for interrupts. Direct IPL interrupts can accidentally violate a critical section created with the AIC master disable. *The use of such IPL interrupts is strongly discouraged.* However, for a programmer, NMI might be useful in a simple debugger that could break out of almost any software situation. The hardware for such a "break" switch would be trivial.

8520 CIAs

The Amiga hosts two 8520 Complex Interface Adaptors (CIA). These adaptors are provided to support the parallel I/O port, keyboard serial interface, serial port control lines, floppy disk controls, game port (mouse) buttons, and several timers.

Both CIA chips include circuitry to generate and control interrupts related to their various functions. Each is capable of generating five internal interrupts emanating from two interval timers, a time of day alarm, a serial port, and a flag input. These

[6] Setting the CPU to level 7 would require much more overhead.

may be separately enabled and disabled through a special control register.

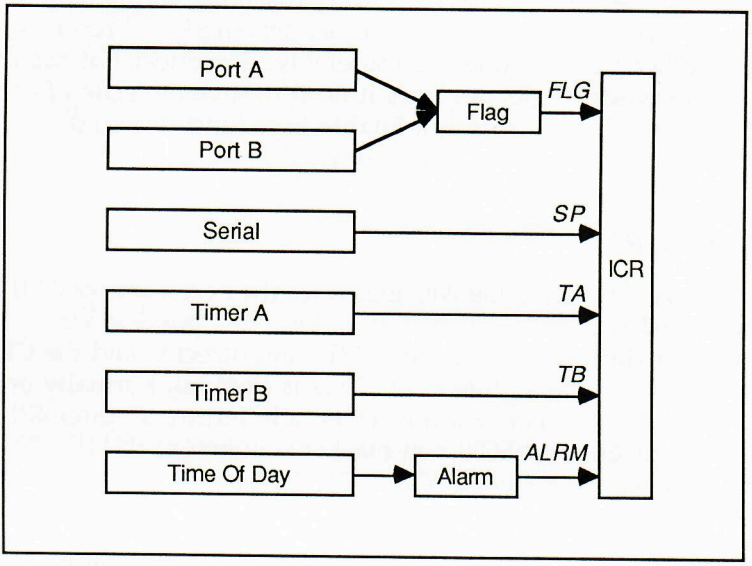

Fig 2.5 – CIA Interrupts

All five internal interrupts are combined to present a single interrupt signal to the AIC. Each CIA requests its interrupt on a different AIC line. One CIA causes a low priority, level two PORTS interrupt. The other causes a high priority, level six EXTER interrupt. These are substantially different priority levels, and this difference permits some degree of flexibility when it comes to timer related CIA interrupts.

The CIA interrupt control register can be read to determine what interrupts are pending. Unfortunately, the 8520's were not designed with multitasking in mind. *Reading the contents of the register clears its value.* This side-effect has created much grief for Amiga programmers. If one handler reads the register, it clears all other pending interrupts – including ones that it may not know how to handle!

Your code should never read the CIA interrupt register directly!

To deal with this hardware limitation, special CIA interrupt handling software had to be written. For each CIA there is an Exec *Resource*. A resource is a body of code similar to a library, but simpler in design. The CIA resources handle the enabling, disabling, setting, clearing, and vectoring of all CIA interrupts.

2.2 CPU

Once the Amiga interrupt controller has made a request on the IPL lines, the responsibility for processing is passed to the 68000 CPU. The CPU goes through a number of steps when such a request occurs, with the end result being the execution of the appropriate processing software.

68000 CPU

As mentioned, the CPU receives encoded interrupt requests from the AIC on three IPL lines. These lines only indicate the priority group of the request, not the exact interrupt.

When an interrupt request occurs, the CPU will delay taking any action until the current instruction has finished execution. During this period the interrupt is said to be *pending*.

The 68000 maintains its current priority level (CPL) as part of the CPU status register. The CPL may vary from zero to seven. It can be manually changed by system software, but it is normally changed automatically as part of interrupt processing. If the requested interrupt priority is greater than the CPL, a new interrupt processing sequence is initiated, otherwise processing is postponed until the CPL is lowered.

Theoretically while waiting for the lower CPL, an interrupt may be postponed indefinitely, but under normal conditions it will only be put off for a small period of time. Because interrupts can nest at different priority levels, the lower the priority of a pending interrupt, the more time it may have to wait.

When an interrupt processing sequence is initiated, the CPU is forced into its supervisor state. At this time the CPL is saved temporarily within the CPU and then raised to match the IPL value of the new interrupt. The pending interrupt has now become *active*.

At this point it should be mentioned that the IPL lines are *open collector* inputs – several output sources may be connected in parallel to the same set of inputs. This allows more than one set of chips to drive the IPL lines[7]. This open collector circuitry

[7] Although, on the Amiga there is usually only one source to the IPL inputs, the AIC.

requires the IPL logic be inverted. When all three lines are high, the IPL is zero (no interrupts requested). If all the lines are low, the IPL is seven (NMI).

Now, the CPU begins to do some magic. The IPL value is placed on the lower address lines of the processor (A1-A3) with all of the other address lines set to one. An address strobe then causes a memory read from the location FFFFFFFN, where the N digit is derived from the IPL value. Essentially, the IPL is being used as an index into a table at the tail end of the ROM.

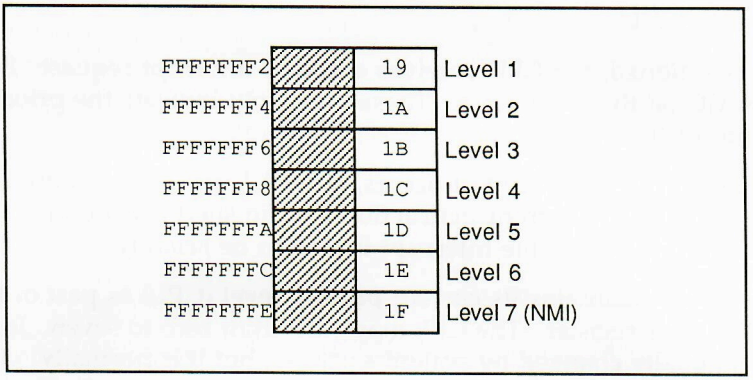

Fig 2.6 IPL Vector Table

From the location addressed (an odd address), a single byte is read[8]. The Amiga requests the 68000 to do automatic vectoring, so this byte is used as a 68000 exception vector number. When shifted left by two (multiplied by four) this vector provides an offset into the 68000 *Auto-Vector* interrupt table. This table contains the entry addresses for Exec's interrupt processing routines.

Once the software has gained control, the CPU automatically generates an interrupt acknowledge on its function code output lines. This acknowledge can be used by external hardware, but the AIC ignores it on the Amiga.

The CPU proceeds to push the program counter (PC) and the status register (SR) onto the supervisor stack[9]. The PC points to the next instruction that would have been executed had the interrupt not occurred. Execution will resume from this address

[8] Note that this operation is invisible to the system software. You can only observe it through the use of a hardware state analyzer.

[9] Other information may also be saved if you are using a 68010 or greater.

at some time in the future. The SR contains the usual 680XX exception information (see below), including the old CPL that was temporarily saved.

Finally, an auto-vector is fetched from the memory table, and control is transferred to an Exec interrupt processing routine.

Auto-Vectors

The auto-vector table is maintained by Exec. The position of the auto-vector table in memory is dependent on the type of processor being used. On the 68000 it is located in the low memory vector space (in chip RAM), starting at 64 hex. For 68010's and beyond, the location of the vector space is determined by the CPU Vector Base register. You should not assume that the auto-vector table is located at 64 hex. The location may change depending on the Amiga CPU.

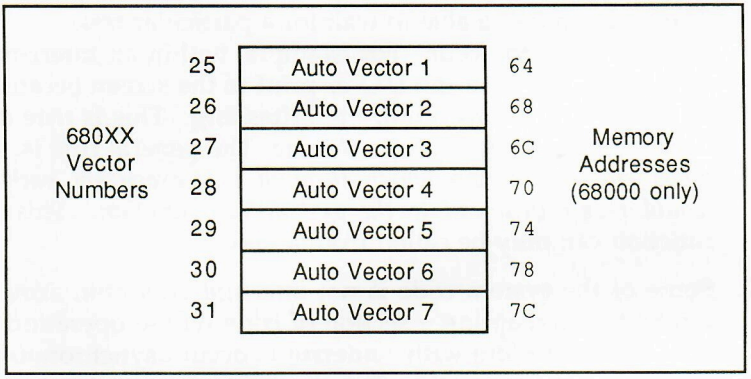

Fig 2.7 – Auto-Vectors

Being in unprotected memory, there is often a temptation to alter the auto-vectors directly. *Unless you are a Master Guru, this action is not advised.* Doing so will steal control from Exec, which performs AIC masking and kernel task scheduling during interrupt processing. Changing the auto-vectors may eliminate any chance of proper system operation.

Supervisor Mode

Interrupts are always processed in 68000 supervisor mode. This mode is considered a privileged processor state. On the Amiga this causes all task processing to be deferred, a special task-independent stack to be used for execution, and access to many system functions to be disallowed. Needless to say, this mode has

a major impact on the form of code you use for processing interrupts.

When executing in supervisor mode, a CPU trap or exception results in a fatal condition. Such an event will arouse the Great Guru and the execution of all tasks and system activities will be immediately halted. This occurs because supervisor mode is considered a *global system state.* No task is in control, so no task can be held responsible when a supervisor exception occurs. If a task had control, Exec could just stop the execution of that task, but when in supervisor mode Exec is forced to shut the system down.

While operating in supervisor mode, you are restricted to using a small subset of the Amiga library functions. There are three reasons for this:

1. Many system functions depend on multitasking. They expect to be executed within a task, not an interrupt. They rely on being able to wait for a particular task relative event to occur. For example, within an interrupt you cannot perform file I/O or print to the screen because both these functions require multitasking. This is true of nearly all device I/O on the Amiga. The general rule is: *never call a function which through any execution path would result in a call to the systemWait function.* This function can only be called from a task.

2. Some of the system code is not *interrupt reentrant.* You might be interrupting a section of code whose operation you could interfere with. Interrupts occur asynchronously to the operation of tasks. The state of task execution may be inconsistent when accessed from an interrupt. For example, variables may not be updated – they may still be in the task's CPU registers. Although many of the Exec kernel functions are protected from being interrupted, some are not. For example, the memory allocation functions are not interrupt reentrant. If you attempt to allocate or deallocate memory from within an interrupt, you put the system at risk.

3. Interrupts should be kept short and to the point. They are not tasks, and should not be used to perform lengthy or indefinite operations. If a more involved operation is needed, use an interrupt to signal a task, and perform the operation there.

From supervisor mode certain CPU registers and operations are more accessible. You can access the system status register, for example, and directly alter the CPU priority level. If not done properly, this will get you into trouble.

Remember to keep in mind that interrupts may be nested: you may be interrupting a lower priority interrupt. Your code cannot rely on the fact that the system was in user mode prior to the interrupt, nor can it rely on returning directly to user mode when it exits. Task scheduling and dispatching also occur from supervisor mode.

Supervisor Stack

Normally, when a task is executing, the CPU is operating with its *user stack pointer* (USP). Whenever the Amiga enters supervisor mode, regardless of the reason, the CPU switches to a separate stack using the *supervisor stack pointer* (SSP).

Supervisor mode is entered for four reasons on the Amiga: exceptions, traps, task scheduling/dispatching, and interrupts. All of these operations use the supervisor stack. They are handled in a sequential stacked fashion, so they may actually be nested on the stack at the same time.

Interrupt processing is probably the most common operation performed on the supervisor stack. When an interrupt occurs the SSP becomes the normal stack pointer. Functions are called and stack is allocated with the SSP. When all interrupt processing has been concluded, the SSP should be at the same location as when the interrupt first occurred.

Within interrupt routines, registers are often saved on the supervisor stack. In addition, interrupt processing code may allocate a small number of temporary variables on the stack. The total stack space used will depend on the functions invoked while processing interrupts. Some system functions use considerable stack space.

As mentioned, the 68000 CPU automatically saves information on the supervisor stack during an interrupt. This information is referred to as a *supervisor stack frame*. This frame will vary between processors being used. The 68000 will have a slightly different format from the 68010, 68020, and 68030. Normally, this is not a problem for interrupt processing code on the Amiga.

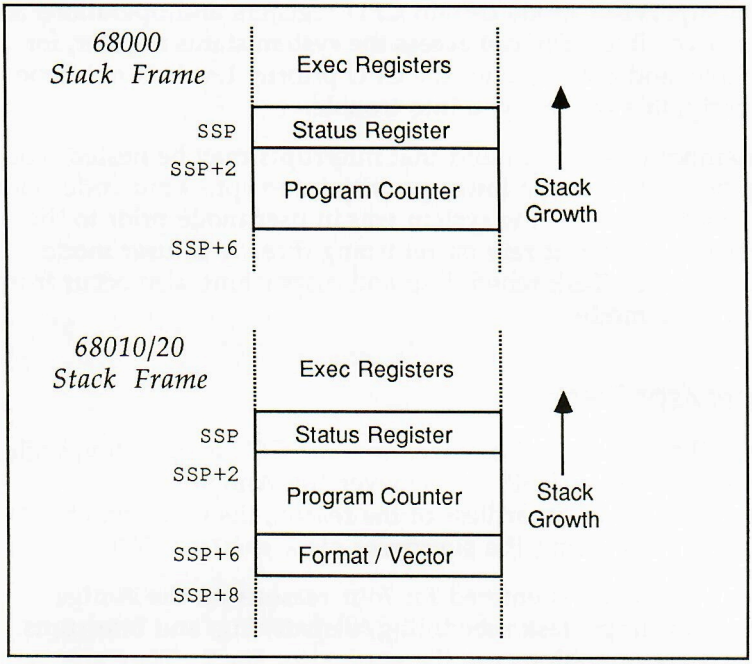

Fig 2.8 – Supervisor Stack Frames

The supervisor stack is created by Exec during startup. Its normal size for V1.2 is 6K bytes. Its upper and lower bounds are recorded in the Exec library data area referenced by ExecBase. For reasons of performance, the supervisor stack is not monitored for stack overflow (although it would be relatively easy to write a program to do so). It is also possible (albeit tricky) to write a program to move the supervisor stack to another part of memory.

Tasks do not use the supervisor stack. Each task has its own stack. Because Interrupts do not interfere with task stack usage, tasks need not allocate the stack space necessary to handle all the possible combinations of interrupts. The size of a task's stack is completely independent of supervisor operations.

CPU Priority Level

Unlike other 68000 systems, on the Amiga the CPU priority level is not normally modified by software. In fact there are only a few places in Exec where the level is altered.

Programmers often want to modify the CPU priorities for two reasons: to disable all interrupts below a certain level (or in some cases all interrupts at all levels); or to allow lower priority

interrupts once a higher priority has gained control. Neither of these are permitted on the Amiga.

To disable interrupts, the AIC should be used, not the CPU. In most cases, the AIC will be able to perform the desired result in less time.

To allow lower priority interrupts from higher priority interrupts could cause either dangerous nesting situations (resulting in stack overflow) or extremely long latency to time sensitive interrupts, so it is to be avoided.

Nested Interrupts

Its been mentioned that interrupts can be nested within each other. This occurs because a higher priority interrupt will take control away from lower priority software.

Figure 2.9 illustrates a possible sequence of interrupt events. Here a task triggers a software interrupt (level 1). Interrupt processing begins, but soon loses control to the vertical blank interrupt (level 3).

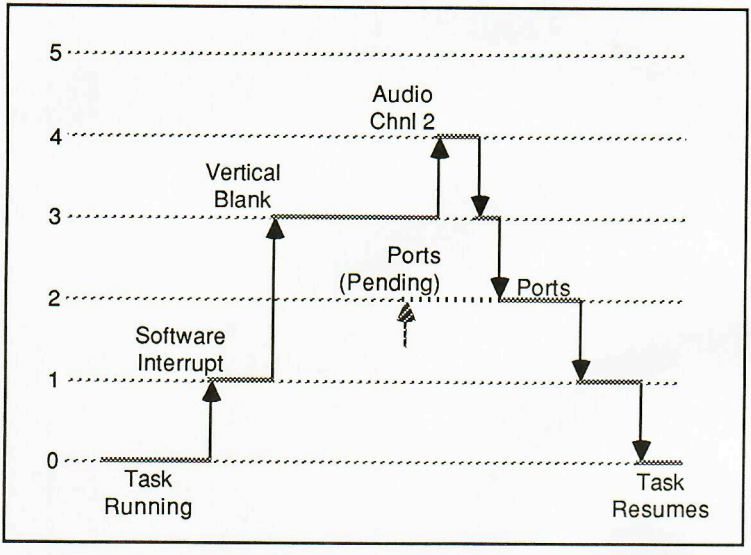

Fig 2.9 – Nested Interrupts

Notice that PORTS (level 2) requests an interrupt, but being at a lower priority it is deferred until the vertical blank interrupt has finished. An audio channel also gets into the act on top of all the

others. Eventually all interrupts have completed and control is passed back to a task (not necessarily the same task).

On the Amiga, the CPU priority level should never be lowered directly by interrupt software. This means that interrupts executing at a given level will never lose control to interrupts of an equal or lower priority, so pseudo-priorities (discussed in the next section) at the same CPU priority will not interrupt each other. Hence, pseudo-priorities do not nest. For example, the blitter will never interrupt vertical blanking even though the blitter has a higher pseudo priority.

2.3 Exec

The Exec library is the master of interrupt control. All interrupts are directed first to Exec which in turn dispatches the individual interrupt routines. Exec performs many operations as the result of an interrupt. Some of these are done before dispatching routines, others are done after the routines have returned.

Spurious Interrupt Rejection

On the Amiga, a spurious interrupt[10] is generated on the average of once every three seconds. These interrupts are usually the result of an interrupt request arising during the first few cycles of an instruction that is disabling an interrupt in the INTREQ register. That is, the request and the disable are occurring at the same time. This happens because a CPU interrupt can be triggered even if the request is removed before the 68000 interrupt acknowledge bus cycle. So, if the AIC presents an interrupt request for at least two bus cycles, the CPU may react *even if the request has been withdrawn.*

Exec does not permit spurious interrupts because they represent an inconsistent state between the INTENA register and the system software. Interrupt processing code may depend on the proper value for INTENA. Exec filters out all spurious interrupts as one of its first operations during interrupt processing, so your code will not be affected by these.

Decoding

As mentioned, there are 14 separate AIC interrupts, but only six auto-vectors[11]. The AIC collects interrupts and generates a CPU interrupt for the priority group.

When an interrupt occurs, Exec uses software to decode the 14 AIC interrupts from the six priority groups. This is done by individually testing each INTREQ bit in the same group. Because

---

[10] Not a 68000 spurious interrupt exception (bus error during interrupt), a different type of event caused by the AIC.

[11] We don't include NMI here, otherwise there would be seven auto-vectors.

there are only a few bits (four at most for the audio interrupts) for each group, this operation does not take much time[12].

Exec tests each of the AIC bits in a predetermined order. The order of testing imposes a *pseudo-priority* on interrupts. This will be discussed later.

Dispatching

Once a particular interrupt has been decoded to a particular AIC bit number, Exec uses this number to reference an ExecBase interrupt vector table. This table contains pointers to both the code and data portions of interrupt processing routines. These routines are called *Interrupt Handlers.*

Before the handler is dispatched, Exec establishes an environment for interrupt processing. Several CPU registers are saved on the supervisor stack. Some of these registers are initialized to predetermined values for use in the handler.

Handling

So far, all of the processing discussed has just been system overhead – nothing "useful" has happened to service the interrupt. The real work begins with the interrupt *Handler.*

Handlers are just subroutines (procedures). They can be created by the system or by the user, and there is only one handler per AIC interrupt. Each handler must completely process its associated interrupt – interacting with related hardware and resetting (clearing) or disabling the interrupt as necessary.

Handlers can be written in a variety of higher level languages. They are allowed to use a limited amount of stack space for local storage, to save registers, and to perform their duties. Most handlers are written in C or assembly code, but others have been created with Pascal, Modula 2, and Fortran.

Handlers are one of the primary topics to be discussed in this document, and we will give many examples of their use in the *Programming* chapter.

---

[12] Of course, it still would have been nice to have this done in hardware.

Sharing

Some interrupts occur on an AIC line that is shared by several pieces of hardware. Other interrupts flag a general sort of event that may be of interest to many tasks. These types of interrupts require some form of interrupt sharing.

For example, it is often desirable to make use of the vertical blanking interrupt to synchronize video events. This same interrupt may also be used by other programs, devices, and libraries at the same time. The graphics library, the input device, the timer device, and several tasks often use the vertical blank interrupt simultaneously.

Table 2.3 identifies the interrupts that are typically shared on the Amiga.

Name	Description
SOFTINT	software interrupt
DSKBLK	disk block done
TBE	transmitter buffer empty
PORTS	I/O ports & external (low)
COPER	graphics co-processor
VERTB	vertical blanking
BLIT	blitter done
AUD2	audio channel 2
AUD0	audio channel 0
AUD3	audio channel 3
AUD1	audio channel 1
RBF	read buffer full
DSKSYNC	disk sync pattern found
EXTER	external (high)
INTEN	master enable (see text)
NMI	non-maskable interrupt

Table 2.3 – Shared Interrupts

Exec provides a simple mechanism for dealing with shared interrupts. It is called an *Interrupt Server*. A server is similar in form to a handler, but many servers can be chained together to process the same interrupt.

When a shared interrupt occurs, control is transferred to a special handler called a *server handler*. This handler interacts with the AIC and dispatches each interrupt server in turn.

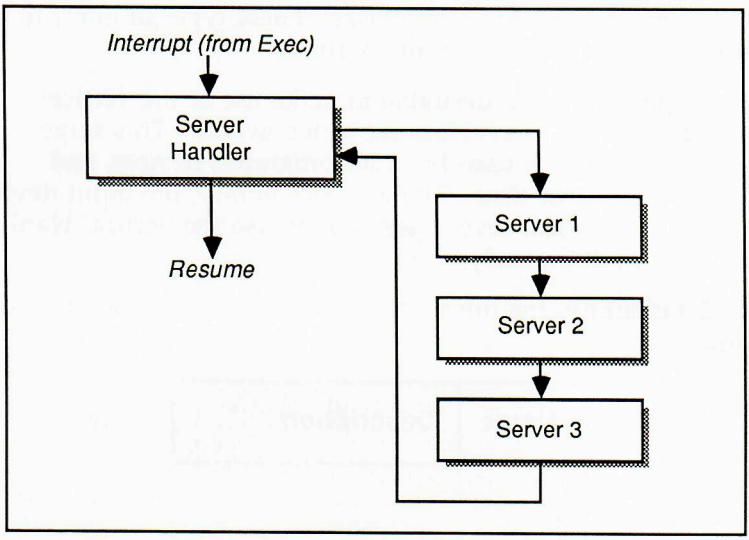

Fig 2.10 – Server Handler and Servers

Servers are chained together by an Exec list structure. They are linked in a priority order – the highest priority being first on the list. Each server is called in turn until the end of the list is reached, or until a particular server decides to terminate the chaining process.

Special external hardware will often have a server attached to the appropriate interrupt chain. The server that will deal with the hardware when an interrupt occurs. Such a server is allowed to terminate the chaining process. This is done to decrease the overhead of interrupt processing for server chains.

When such a server is executed, it determines whether its hardware is the source of the interrupt. If it is, the server processes the interrupt, and terminates the chain by returning a special value. If the interrupt was not caused by its hardware, it returns a different value, and control is passed to the next server.

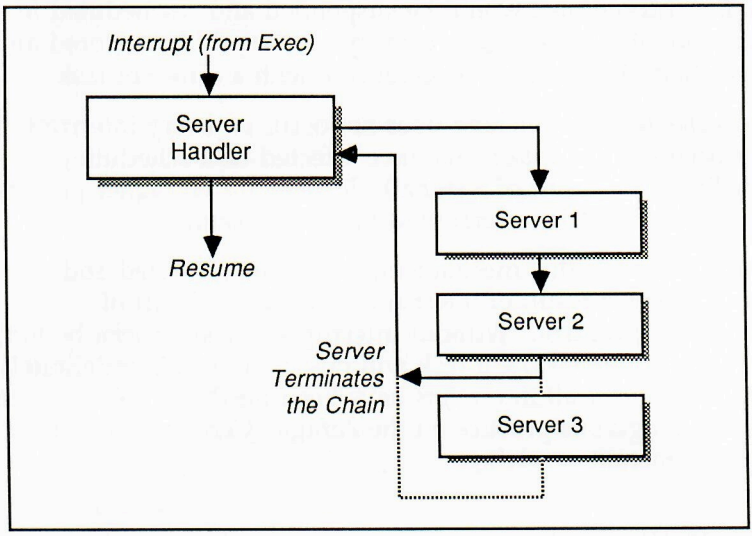

Fig 2.11 – Server Chain Termination

Take a hard disk device as an example. Let's say that the hard disk controller signals a PORTS interrupt. Control is passed to each server in the PORTS chain, until the hard disk server is executed. This server notices that its device is the source, and processes the interrupt. It then returns with a flag to indicate that the PORTS interrupt chaining should be stopped.

The same Exec code supports all server handlers. It is installed in the system during startup. More details about servers will be provided in the programming chapter.

Task Dispatching

When interrupt processing has been completed, Exec must determine where to resume. If the interrupt occurred while the CPU was in supervisor mode, Exec simply returns control via the current stack frame. Supervisor mode implies that the interrupt did not happen within a task. Execution will continue at the next instruction following where the interrupt occurred.

However, if the interrupt happened while the CPU was in user mode, Exec has some work to do. User mode indicates that a task was executing prior to the interrupt. Before resuming the task, Exec must determine whether a higher priority task is awaiting execution. This would be the case if an interrupt sent a signal to a higher priority waiting task. The signal would cause the task to be readied for execution. The Exec scheduler would be invoked.

The interrupted task would be suspended and rescheduled for execution later. The higher priority task would be restored and dispatched. Execution would resume with a different task.

Of course, task dispatching does not occur on every interrupt. It only occurs when interrupts have affected task scheduling (usually as the result of a signal). If there are no higher priority tasks to execute, the interrupted task will resume.

Notice that with this mechanism, tasks are scheduled and dispatched as a result of interrupts. This is the heart of multitasking control. Without interrupts, a task cannot be forced to relinquish the CPU – a task will possess the CPU indefinitely. This can occur if all interrupts have been disabled. We warn that this is a dangerous practice on the Amiga. Certain system tasks require constant servicing.

Pseudo-Priorities

As we mentioned in the **Decoding** section above, Exec uses software to determine which interrupts have occurred within a priority group. This is done by testing INTREQ one bit at a time. If a bit is set, the appropriate handler is dispatched.

This selection process imposes an ordering to interrupts. Because one interrupt may in some cases take precedence over another, a form of prioritization is created. In Exec this is called *pseudo-priority*.

Of course, pseudo-priorities are not really priorities at all. They are an artificial type of priority. At the same CPU priorty level pseudo-priorities will not interrupt each other.

Table 2.4 gives the complete picture for AIC interrupts and their priorities. The second column is the interrupt pseudo-priority. The third column indicates the AIC bit number.

This table is organized in an increasing priority order. Notice that the first three entries appear in a different order than when listed by AIC bit numbers.

Pri	P	B	Name	Description
1	1	2	SOFTINT	software interrupt
	2	1	DSKBLK	disk block done
	3	0	TBE	transmitter buffer empty
2	1	3	PORTS	I/O ports & external (low)
3	1	4	COPER	graphics co-processor
	2	5	VERTB	vertical blanking
	3	6	BLIT	blitter done
4	1	9	AUD2	audio channel 2
	2	7	AUD0	audio channel 0
	3	10	AUD3	audio channel 3
	4	8	AUD1	audio channel 1
5	1	11	RBF	read buffer full
	2	12	DSKSYNC	disk sync pattern found
6	1	13	EXTER	external (high)
	2	14	INTEN	master enable (see text)
7	1	15	NMI	non-maskable interrupt

Table 2.4 – Pseudo-priorities

2.4 Software Interrupts

The term *Software Interrupt* means different things in different operating systems. The term as it is used on the Amiga, refers to a type of system interrupt that is generated by a software event. These interrupts operate in an environment that is nearly identical to regular hardware interrupts.

For those readers familiar with other types of operating systems it is important to note that Amiga software interrupts are not task based interrupts. They are system global interrupts. They differ from hardware interrupts because they originate from software events, not hardware events. On the Amiga, asynchronous task interrupts are referred to as *Exceptions*, and they will be discussed in the next section.

Software interrupts are a type of "micro-task" on the Amiga. They are simpler than a regular task and are easier to create. They are often used to unload duties from higher priority interrupts, allowing them to defer time consuming operations to a lower priority. This makes the system more responsive to lower level interrupts.

Software interrupts are used within the Amiga system itself. For example, the performance of task rescheduling was improved dramatically by employing software interrupts.

Environment

Software interrupts (SWI) execute in a normal interrupt environment: supervisor mode on the supervisor stack. They are not related to tasks in any way. The same restrictions that limit access to system functions during hardware interrupts, also apply to SWIs.

Software interrupts also use the identical Exec interrupt data structure. So if you know how to setup and process hardware interrupts, you know most of what you need for SWIs.

Although software interrupts are triggered by software only, they are supported with the Amiga hardware. The Amiga interrupt controller reserves a level one interrupt exclusively for software interrupts. It turns out that this was not a necessity, but it does improve software interrupt performance considerably.

Priorities

Within the general scheme of things, software interrupts operate at a lower priority than hardware interrupts, but a higher priority than tasks. All pending hardware interrupts are processed before any SWIs, and all SWIs are processed before tasks. SWIs execute in supervisor mode at CPU priority level zero.

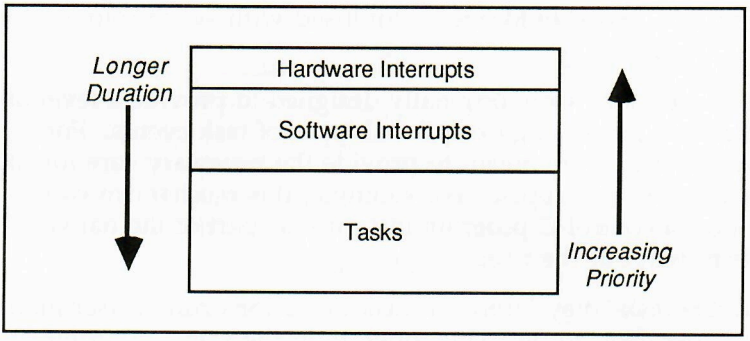

Fig 2.12 – Execution Priorities

Software interrupts have their own priority system. The highest priority pending SWI is always the first dispatched. Unlike hardware interrupts, SWIs cannot really be nested. However, when a SWI is finished, the system will run the highest priority pending SWI next. This means that an SWI can invoke a higher priority SWI.

Requesting

There are two ways to request a software interrupt: directly with an Exec system function, or indirectly when an intertask message arrives. We will describe how to perform both of these in the *Programming* chapter.

2.5 Task Interrupts (Exceptions)

Exec provides another type of interrupt that is specific to tasks rather than being system global. In Exec's terminology *Task Exceptions* are a form of interrupts that occur within the environment of a task. Exceptions are full fledged asynchronous interrupts, and should not be confused with synchronous CPU exceptions[13].

Task exceptions were originally designed to provide a level of control for processing exceptional types of task events. For example they were meant to provide the necessary core for high quality user interrupts. For example, this mechanism could support a control-C program interrupt as part of the native system without the need for polling.

Like the tasks they interrupt, task exceptions run in user mode, not supervisor mode. They operate in the same environment as the interrupted task and use its stack.

When a task exception occurs, the task is immediately forced into a special exception state. The current CPU registers are saved in the task's stack. The task exception handler is then invoked as a sort of "super-task". The handler can freely manipulate the state and registers of the task as if it were running in an independent CPU. When the handler has finished, the task may be resumed where it left off or control can be transferred elsewhere.

Bad News

Unfortunately, task exceptions have a limited use on the Amiga. Although they are supported internally within Exec, none of the system functions support them properly. An exception may occur deep within a system function, in a context unfamiliar to the exception handler. In such cases, the handler may not be able to adequately "unwind" the current environment, or the system may recurse on itself in unexpected ways.

[13] Address error, bus error, illegal instruction, zero divide, etc.

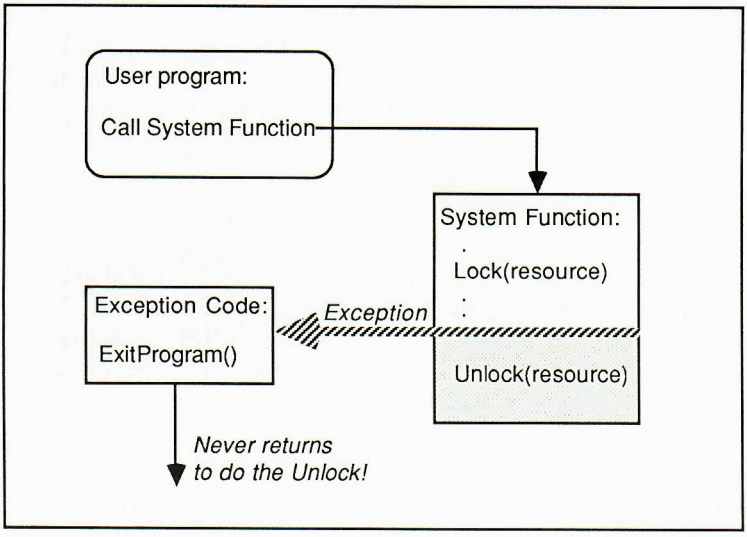

Fig 2.13 – Task Exception Problem

Most of the problems occur when it comes to owning or allocating system resources. Suppose you called a graphics function which asked for control of the blitter. If a task exception yanked you out of this code, the blitter may never again become free. A similar problem could happen with memory allocation.

This means that task exceptions should only be used:

1. When they do not divert the flow of control permanently. You can use them to make very simple state changes though – like altering a memory location.

2. When your program does not call other system functions. (This really limits the possibilities; however, it may be possible to write a device driver where the gain is greater than the loss).

3. If you can disable the exception before calling system functions, then re-enable and query when returning. This could be done by providing a special set of "filtered" libraries for the system.

Task exceptions will not be discussed further in this issue of the *Guru's Guide*. Perhaps a later issue could address this topic in more detail.

3 - Programming

This chapter develops the interrupt theories from the previous chapter into a programming practice. Here the focus is on the data structures, system functions, and programming techniques needed to process Amiga interrupts. This chapter also includes several working examples.

3.1 Languages

Your choice of a programming language will greatly affect the way you write interrupt processing code. Languages like C, Modula 2, Pascal, and Assembly have minimal impact on the structure of interrupt code. Other languages like BASIC, BCPL, Lisp, and Logo may not be able to handle Amiga interrupts in a direct fashion. Several factors should be considered when choosing a programming language:

Environment
: Can you properly restore the language's run-time environment within an interrupt processing routine, and is it efficient to do so? Some languages, especially interpreters, require complex environments which require many processor cycles to restore.

Registers
: Does it save the CPU registers properly? Languages make different assumptions about scratch registers and will not automatically save them.

Functions
: Does the language have a type of function, subroutine, or procedure that is similar to native 68000 form? Can you create a pointer to

it? Does it perform an RTS when exiting? Can it set the 68000 zero condition code?

Stack Does the language use a minimum amount of stack? It should use as little as possible. Does it perform stack checking? Interrupts run on the supervisor stack, so the stack bounds may be different than the language thinks. Does the language require frame pointers? What would happen if it tried to walk back its frame pointer chain? Some languages use their own special stacks. Can they interface to the system for interrupt processing?

Reentrant Is the language reentrant? Will it be in a consistent state when an interrupt occurs? Many languages keep active environment information in CPU registers making it inaccessible during interrupts.

Exceptions Does the language perform its own exception handling? If so, can it be disabled during interrupt processing? Languages with try/recover, catch/throw and signal/exception mechanisms could cause problems.

Background Does the language make invisible calls to system functions? Some languages may perform periodic polling of devices, timers, etc. These could lead to external I/O or multitasking – both of which would be disastrous from interrupts.

Of course, even if your favorite language does not meet these requirements, it still may be able to process interrupts *indirectly*. Some interrupts do not need to be processed within the system's interrupt environment – many just forward processing back to a task via a signal. Your language may be able to deal with task signals rather than interrupts. Other languages may let you load assembly code object modules for processing interrupts.

Of course on the Amiga, C and Assembly have long been the favorites for system programming. Most of the Amiga kernel and many of the utilities were written in C. The higher performance modules of the kernel were written in Assembly. Because of this, the examples provided in this chapter will be written mostly in C with a few simple optimizations in Assembly.

C Language

C can best be though of as a *transportable assembly code.* It is widely available, relatively portable, and quite flexible. Programmers like it because it is performance oriented – they "know" what kind of code it is generating. It is a workhorse.

C has an ugly side however. It is a Dr. Jekel and Mr. Hyde type of programming language. It is diabolically easy to get yourself into trouble, and C code is often difficult to debug. It can provoke you into fits of rage, or send you to an asylum. Using C will definitely cut your life expectancy by ten years or more. If you are not careful, it will compound the problems encountered in interrupt processing.

The C run-time environment is quite simple. Most C's use one or two CPU registers for their environment pointers and have a register convention similar to that of the Amiga system software.

C language functions are native 68000 functions. They use the RTS instruction and they can set the CPU's zero condition flag before exiting.

C utilizes the stack in a straight forward fashion for variable storage, parameter passing, and function frame linkage. Stack overflow checking is a compiler option. C does not have built-in exception handling, nor does it call background functions during normal execution.

Some C compilers provide optional *small memory models* to improve performance and reduce the code size. Such compilers reserve a single CPU register as the base for a data segment[1]. When used in interrupts, this register must be restored before accessing global variables.

Some C compilers provide functions to assist interrupt processing. For instance, the *Manx Aztec C* compiler provides two functions: **int_start** and **int_end**. The start function saves additional scratch registers that will be used by the compiler. It also restores the data segment registers when the small memory model is being used. The end function restores all saved registers. An interrupt routine written in Aztec C would look like:

[1]The data segment may also contain vectors to functions as well.

```
IntrProc()
{
    int_start();
    ... do something useful...
    int_end();
}
```

Assembly Language

A direct way of handling interrupts is to program in 68000 Assembly code. This route offers a number of advantages over C but also has its drawbacks.

Assembly code offers much greater control of the processing situation. It has no inherent operating environment. Nothing is done behind your back. The programmer makes all decisions regarding the way the CPU registers and the stack will be used.

Well written assembly code can execute considerably faster than the best compiled C code. Experience shows that C conversion to Assembly results in twice the speed, and designing an Assembly routine from scratch will often yield greater than three times improvement[2]. Assembly code is also more compact in size. It is often less than one third the size of the equivalent C code.

Of course if you are writing a small, simple program, the speed of execution is often irrelevant. If an operation takes eight microseconds in C and only three in Assembly, but the operation is only executed every 16,000 microseconds, converting to Assembly would not be worth it.

On the down side, programming in Assembly may take more time. Even if you are a highly experienced coder, there are many details to think out: register utilization, parameter passing, looping constructs, comparisons, etc. Macros can help save a small amount of this drudgery, but not all of it.

Combination

Sometimes a combination of C and Assembly is best. Use C for the bulk of your code and Assembly where performance is important. You may decide to write your interrupt code in Assembly, but the rest of your program in C. Fast interrupt code improves the performance of not only your program, but the system overall.

---

[2]When coding from scratch, a programmer takes advantage of addressing modes and register utilization beyond that found in modern C compilers.

3.2 Data Structure

To program interrupts, there is only one data structure of importance: the **Interrupt**. Handlers, servers, and software interrupts all use this same structure. It is quite simple – both easy to learn and use.

The **Interrupt** structure is an Exec **Node** structure followed by two long words. The first long word points to an arbitrary data segment. The second long word points to an interrupt processing routine.

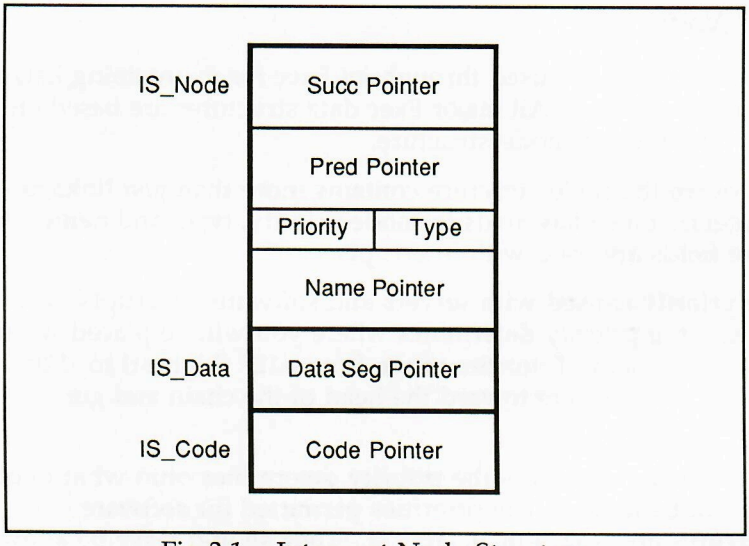

Fig 3.1 – Interrupt Node Structure

The definition for this structure can be found in the include file directory for Exec. The C code definition is found in the *interrupts.h* file:

```
struct Interrupt
{
    struct Node is_Node;
    APTR is_Data;
    VOID (*is_Code)();
}
```

The **APTR** (absolute pointer) data type refers to any location in memory. The **VOID** line establishes a pointer to a function that does not return a result.

The assembly code equate definition is found in *interrupts.i*:

```
STRUCTURE     IS,LN_SIZE
    APTR      IS_DATA
    APTR      IS_CODE
    LABEL     IS_SIZE
```

The **STRUCTURE, APTR,** and **LABEL** words are all assembly macros. Their primary purpose is to set the field offsets within the structure. The first line indicates that the structure prefix is called **IS** and its first data field is located at an offset of **LN_SIZE** bytes. Each **APTR** adds an additional four bytes to the offset. The last line equates **IS_SIZE** with the total size of the structure.

List Node

Exec list nodes are used throughout Exec for maintaining lists of system structures. All major Exec data structures are based on this fundamental node structure.

Of course the node structure contains more than just linkage pointers. It also has fields for node priority, type, and name. All these fields are used with interrupts.

The priority is used with servers and software interrupts. For servers, the priority determines where you will be placed within the server chain. Priorities range from +127 (highest) to -128. Higher priorities are toward the head of the chain and get serviced first.

For software interrupts the priority determines onto what queue you will be linked. The priorities permitted for software interrupts are: +32, +16, 0, -16, -32. They should *never* be allowed to range outside +32 to -32, as this could crash the system. Values between these will be rounded down (e.g. 31 becomes 16, and -31 becomes -32).

The name field is used to identify the interrupt. It points to a zero terminated ASCII string of characters. The importance of a name will become clear. You will want to give a good name to your interrupt.

The type field specifies the Exec structure type. A numeric constant **NT_INTERRUPT** indicates that the structure is an interrupt. This constant is defined in the *nodes.h* and *nodes.i* include files. It is used for all types of interrupts.

Important note: When queued to execute, a software interrupt structure has its node type changed automatically to

NT_SOFTINT. This is done to prevent it from being linked more than once. When the interrupt has been removed from its queue immediately before it is executed, its type field is restored to **NT_INTERRUPT**.

Data Pointer

ROM based code often allocates its memory dynamically. This can create a problem when it comes to interrupts: ROM code cannot assume the location of its data. Direct references to absolute memory are not allowed on the Amiga, so ROM based interrupts have no easy way to access dynamic data segments.

The **IS_DATA** field exists to solve this ROM code problem. Kickstart and expansion hardware ROM's use this field for locating their data.

Even though many of the interrupts on the Amiga are not derived from ROM code, this field is still a handy place to store a pointer to relevant data. In fact, any value can be stored in this field. Some programs actually store data in this field rather than a pointer.

Code Pointer

The **IS_CODE** field of the interrupt structure is a pointer to the entry point of an interrupt processing routine. This routine is called as if it were a subroutine, and it should return appropriately. It will be explained in more detail shortly.

Initializing

There are two common techniques for initializing an **Interrupt** structure. The structure can be defined within a program as a pre-initialized structure, i.e. all of its fields defined statically. The alternative is to create the structure dynamically during execution. Both of these techniques can be done from C or Assembly. The former is simpler for quick experimental programs. The latter gives somewhat more flexibility at the expense of larger code.

Here is an example of a statically built C interrupt structure. In this case, the interrupt data segment is only a single long word. For the interrupt name field, a normal C character string is supplied.

```
#include <exec/interrupts.h>

VOID    IntrProc(); /* interrupt processing code */
long    IntrCount;   /* vertical blank counter    */

struct Interrupt VertBlank =
{
    NULL,                   /* node pred*/
    NULL,                   /* node succ*/
    -60,                    /* node pri */
    NT_INTERRUPT,           /* node type*/
    "example.interrupt",    /* node name*/
    (APTR)&IntrCount,       /* data ptr */
    &IntrProc               /* code ptr */
};
```

Notice that the data pointer requires an APTR type coersion to make the C compiler happy.

The following function creates and initializes an interrupt structure dynamically. This example is a bit more complex because it involves memory allocation. When memory is requested with **AllocMem**, it is directly allocated from system memory, not from your language's pool. This will not be a problem as long as the memory is freed when no longer needed[3].

```
#include <exec/interrupts.h>
#include <exec/memory.h>

struct Interrupt *MakeIntr(name,pri,code,data)
    char *name;
    int  pri;
    VOID (*code)();
    APTR data;
{
    extern struct Interrupt *AllocMem();
    register struct Interrupt *intr;

    intr = AllocMem(sizeof(*intr), MEMF_PUBLIC);
    if (intr == NULL) return NULL;

    intr->is_Node.ln_Pri = pri;
    intr->is_Node.ln_Type = NT_INTERRUPT;
    intr->is_Node.ln_Name = name;
    intr->is_Data = data;
    intr->is_Code = code;

    return intr;
}
```

[3] This can best be done by writing a companion function that does deletion.

There is nothing unusual about this function. It returns either a pointer to the interrupt or a null (when it fails to allocate the structure).

Notice that the structure is allocated with the **MEMF_PUBLIC** flag. This is done because interrupt structures belong to the "system" class of memory. Using this flag will allocate the memory from the global system space (when and if such a space is ever made possible through the use of a memory management unit).

Clean-up

The structure allocated with **MakeIntr** must be freed back to the system pool before the program exits. If this is not done, the memory will be stranded – never again available for use.

```
#include <exec/interrupts.h>

FreeIntr(intr)
    register struct Interrupt *intr;
{
    if (intr == NULL) return;
    if (intr->is_Node.ln_Type != NT_INTERRUPT) return;

    intr->is_Node.ln_Type = 0;

    FreeMem(intr, sizeof(*intr));
}
```

This function includes two simple safeguards. First, it ignores NULL interrupt pointers. Second, it supports a crude means to help prevent deletion from occurring twice. The node type is used as a flag to indicate that the structure has been freed. Of course this is not foolproof because the memory is free for reuse and it may get set to the **NT_INTERRUPT** value again. You may decide to delete it.

3.3 System Functions

Interrupts are supported with several Exec library functions. These functions primarily provide a means of connecting and disconnecting interrupts to the system:

SetIntVector	establish a new interrupt handler
AddIntServer	add a new interrupt server
RemIntServer	remove an interrupt server
Cause	generate a software interrupt

These functions are documented in the *ROM Kernel Manual*. They are described in the RKM *Exec* volume, and their function headers are printed in the RKM *Libraries and Devices* volume. Unfortunately, they are not well explained in either volume, so the following pages describe these functions in more detail.

SetIntVector

Summary

SetIntVector – establish a new interrupt handler

Invocation:

```
SetIntVector(AICBit,  IStruct)
```

Accepts:

```
int  AICBit    (D0)
struct  Interrupt  *IStruct    (A1)
```

Returns:

```
struct  Interrupt  *    (D0)
```

Arguments

AICBit is an integer used to indicate the interrupt vector to be set. It is the interrupt bit number as found in the AIC registers. It is restricted to the range of 0 to 15. The include files *intbits.h* and *intbits.i* in the *hardware* directory contain predefined constants for use with this function. They are of the form **INTB_***NAME*.

IStruct is a 32 bit pointer to an interrupt data structure. The structure should be properly initialized prior to calling this function.

Return

This function returns a pointer to the interrupt structure previously processing this interrupt. If the interrupt was not being processed, a zero is returned. See discussion below for fine points.

Description

This function informs the system to direct all occurrences of a particular interrupt to a specified interrupt handler. The handler is responsible for processing and clearing the interrupt.

This function should *not* be used with server interrupts. To do so would steal control from the system server handler, thus preventing the normal execution of the server chain.

This function does not protect against you specifying the wrong parameters. Doing so may provoke the Guru. This function inserts your interrupt structure into a book-keeping list, so take care not to use it more than once with the same structure.

This function has no affect on the state of the related interrupt. It does not automatically enable, disable, set, or reset the interrupt. These operations should be performed at the proper times by your code. Usually it is best to disable the interrupt before calling this function – then enable it when it is needed.

Your handler code will need to clear the correct AIC INTREQ bit in order to reset the interrupt.

When you have finished using the interrupt, you should call this function with a NULL as the structure pointer. This will remove your structure from Exec.

Issues

This function has the ability to steal interrupts away from other programs. When the interrupt is already in use, calling **SetIntVector** replaces the current handler with your new handler. A pointer to the old handler's interrupt node is returned. This can create a difficult problem.

Exec assumes that if you are stealing an interrupt vector, you are doing so at the request of the user (he ran your program or whatever). It would be proper procedure to inform the user that you have taken control of this interrupt away from some other program. Now the importance of naming your interrupt nodes becomes clear. This name may be presented to the user in a message:

```
WizDisk.device just took interrupt
control from TrackDisk.device
```

Of course you'll have to decide when such a message is appropriate. Some users will frown on such messages.

When you have finished handling interrupts, you should call **SetIntVector** again to remove your structure from the system. In some cases you may want to replace the structure you received back from the original call. This is problematic. The program you displaced may or may not still be operating. Unless you know exactly what you are doing, it is not advisable to replace the previous structure. It can be done if it was one of your structures in the first place, or if it is a piece of code you understand quite well. For example, if you know that the TrackDisk device has not left memory, you might succeed in restoring its interrupts. However, you will need to consider the state of its interrupt enables as well. You could save these along with its structure pointer.

Internals

Internally, this function copies the data and code pointers from the interrupt structure into Exec's interrupt vector table. This is done to reduce latency when the actual interrupt occurs. Exec needs only to index into its table, it does not need to use the interrupt structure.

It saves the location of the interrupt structure in two places. It deposits the pointer in the vector table itself and links the node into a book-keeping list. This latter action is somewhat redundant and was probably unnecessary, but it was done to provide consistency with the rest of Exec.

When a zero is supplied for the structure parameter, the previous structure will be unlinked, and the vector table entries cleared. The vector table's data and code entries are actually set to -1 to cause an address error (hence a crash) should the interrupt accidently remain active.

AddIntServer

Summary

AddIntServer – add a new interrupt server

Invocation:

```
AddIntServer(AICBit,  IStruct)
```

Accepts:

```
int  AICBit    (D0)
struct  Interrupt  *IStruct    (A1)
```

Arguments

AICBit is an integer used to indicate the server chain to link to. It is the interrupt bit number as found in the AIC registers. It is restricted to those interrupts for which there is a server handler. The include files *intbits.h* and *intbits.i* in the *hardware* directory contain predefined constants for use with this function. They are of the form **INTB_***NAME*.

IStruct is a 32 bit pointer to an interrupt data structure. The structure must be properly initialized prior to calling this function.

Description

This function adds your interrupt server to the chain of servers for a particular interrupt. The priority of your interrupt node will determine its placement in the server list. Higher priority servers get processed first.

When the first interrupt server is added to the list, the interrupt will automatically be enabled. Your code might be called immediately after being added, so it must be ready to execute.

If you have no preference as to the priority of your server, a priority value of zero should be used.

The system server handler takes the appropriate action when it comes to resetting the INTREQ AIC register. A server should not reset this register.

Specifying interrupt 15 adds your structure to the NMI server chain. Because this is not an AIC interrupt, many restrictions apply to NMI interrupts. See the *Principles* chapter for more details.

AddIntServer does not protect against you specifying the wrong parameters. Doing so may provoke the Guru. It should *not* be used with handler interrupts.

When your program has finished using an interrupt it should remove its structure from the server list by calling the **RemIntServer** function (below).

AddIntServer inserts your interrupt structure into a server list. Because of this, you must never call **AddIntServer** more than once with the same structure without first removing the structure with **RemIntServer**.

Issues

With shared interrupts, control propagates from one server to the next. A server can terminate this flow of control by returning with the CPU zero flag set.

This was not a wise choice for a termination flag. First of all, if you are using a higher level language, you must find some way to set this flag. In most C compilers an operation like **return 0** is enough to set the zero flag. But this brings us to a second problem. *The zero flag must remain set!* This is more difficult. Higher level languages usually clean up and restore registers before they return. It is possible that they could inadvertently clear the zero flag during such operations. For example, some compilers may optimize a single register **MOVEM** instruction to a **MOVE**. **MOVEM** does not affect the zero flag, whereas **MOVE** will! *This can be the source of an intermittent premature termination of the server chain!* It could seriously affect the proper functioning of certain system functions, and it is a difficult bug to detect. It is not only a problem with compilers, some assemblers perform the same optimization.

You have a couple of ways to help prevent this problem.

- Know exactly what code your compiler is generating.
- Turn off compile or assembly options that perform **MOVEM** to **MOVE** optimizations.
- Add a **MOVEQ #0,D0** or a **MOVEQ #1,D0** instruction to the tail of your function.

There is also a problem that can occur when adding a server to the **VERTB** chain at a priority higher than 10. The graphics library interrupt server (at priority 10) incorrectly assumes that the **A0** register contains a pointer to the custom chips. If

your server runs at a higher priority, it will need to exit with the correct value in **A0**. Do this by saving and restoring **A0**, or by setting it to **Custom** before returning.

Internals

This function disables interrupts momentarily while it adds your interrupt node to the server list. This prevents a possible inconsistent list structure.

RemIntServer

Summary

RemIntServer – remove an interrupt server

Invocation:

```
RemIntServer(AICBit,  IStruct)
```

Accepts:

```
int  AICBit    (D0)
struct  Interrupt  *IStruct    (A1)
```

Arguments

AICBit is an integer used to indicate the server chain to unlink from. It is the interrupt bit number as found in the AIC registers. It is restricted to those interrupts for which there is a server handler. The include files *intbits.h* and *intbits.i* in the *hardware* directory contain predefined constants for use with this function. They are of the form **INTB_***NAME*.

IStruct is a 32 bit pointer to an interrupt data structure to be removed.

Description

This function removes your interrupt server from the chain of servers for a particular interrupt.

This function does not protect against you specifying the wrong parameters. Doing so may provoke the Guru.

This function removes your interrupt structure from a server list. You must never call **RemIntServer** on a structure that has not been added with **AddIntServer** first.

When the last interrupt server is removed from the list, the interrupt should automatically be disabled (see comment below).

Issues

RemIntServer for release 1.2 has a bug. It does not properly disable the interrupt when the last server is removed. It is possible for the interrupt to be left on!

Theoretically, under normal operating conditions, the **RemIntServer** will affect the following server interrupts:

3 PORTS: not a problem, CIA server is usually present.

4 COPER: will not disable Copper interrupt, but causes no other problems. If Copper list is deleted than the interrupt shouldn't occur.

5 VERTB: not a problem, Graphics and Timer servers are usually linked.

13 EXTER: not a problem, CIA server is usually present.

15 NMI: not a problem, nothing to disable anyway.

So for most interrupts, you don't need to worry about this bug. However, if it does somehow affect your code, there are a two ways to fix the problem:

• You can use **SetFunction** to patch the Exec library. A degree of care is required to do this. A special "resident patch" program should be used, not your application[4]. Also, the Exec library version number should be checked before making any changes.

• You can perform the equivalent operation of **RemIntServer** in your own code.

Internals

This function disables interrupts momentarily while it removes your interrupt node from the server list. This prevents a possible inconsistent list structure.

[4] There is no safe way to "unset" a function. Some other program could have set the function after your change.

Cause

Summary

Cause – generate a software interrupt

Invocation:

```
Cause(IStruct)
```

Accepts:

```
struct  Interrupt  *IStruct    (A1)
```

Arguments

IStruct is a 32 bit pointer to an interrupt data structure. This should be properly initialized for a software interrupt. The node type, **LN_TYPE**, must be set to **NT_INTERRUPT**.

Description

This function generates a software interrupt (SWI). The input specifies which SWI is to occur.

If this function is executed from user mode with interrupts enabled, the SWI will occur immediately. If executed in supervisor mode, the interrupt will execute after all other interrupts have finished, but before control is returned to the current task.

Five priority levels are available for software interrupts: +32, +16, 0, -16, -32. The priority is set with the node priority byte, **LN_PRI**. *Never* use a priority outside the +32 to -32 range. Priorities are not checked by the system.

Unlike other list related functions, SWIs are protected from being linked more than once. The **Cause** function can be called with the same software interrupt multiple times without harm. This gives SWIs a behavior similar to task signals.

Issues

Software interrupts (SWI) execute in supervisor mode at level zero, so they can be interrupted by any other hardware interrupt.

Internals

This function temporarily modifies the node type in the interrupt structure. The type is set to **NT_SOFTINT** once it

has been queued. This prevents all attempts to link the structure again.

3.4 Monitor

When working on system code, it is often helpful to see what's happening. Investing a few minutes to create special tools can be well worth the effort.

The program in this section will provide you with a simple tool for printing out a table of Exec's interrupts. When executed from the CLI, the program produces output similar to this:

```
Guru's Guide Interrupt Table Printer

IV E INTR     P S   CODE     DATA    NPR NAME
-- - -------- - - -------- -------- --- --------------
 0   TBE      1   0003642e 0000b73c     serial.device
 1   DISKBLK  1   00fc4a80 00001f76     (missing)
 2 * SOFTINT  1   00fc1380 00000000
 3 * PORTS    2 * 00fc461c 00001b10   0 ciaa.resource
 4 * COPER    3 * 000323ac 00000000   0 copper.example
 5 * VERTB    3 * 00fc6cdc 00002296  10 (missing)
                * 00fe5cc8 00002dec   0 gameport.device
                * 00fe96da 00002fee   0 timer.device
 6   BLIT     3   00fc6d8e 00002296     (missing)
 7   AUD0     4   00fc35a4 0000313e     audio.device
 8   AUD1     4   00fc35a4 000031aa     audio.device
 9   AUD2     4   00fc35a4 00003216     audio.device
10   AUD3     4   00fc35a4 00003282     audio.device
11   RBF      5   00036190 0000b73c     serial.device
12   DSKSYNC  5   00fc4a98 00001f76     (missing)
13 * EXTER    6 * 00023666 00000000  20 stickydisk.device
                * 00fc4610 00001ed8   0 ciab.resource
14 * INTEN    6
15   NMI      7 *
```

The second column indicates whether the interrupt is enabled in **INTENAR**. The fifth column is marked for servers.

There is a limitation in this program that you should know. It assumes a static Exec environment – it makes no allowances for structures that may be in flux. For example, if an interrupt were removed from a server chain while being accessed by this program, the output could be wrong, or worse: the program could crash. This problem can be solved, but for your own use as a tool, it isn't worth the effort.

```
/*****************************************************/
/***  Guru's Guide Example: Interrupt Monitor    ***/
/*****************************************************/

/* NOT FOR GENERAL USE: does not protect against  */
/* changes while accessing system structures!     */

#include <exec/exec.h>
#include <exec/execbase.h>

#define REG          register
#define FIRST(n)     (struct Interrupt *)((n)->lh_Head)
#define EMPTY(n)     ((n)->lh_Head == &n->lh_Tail)
#defineNEXTINT(i)    (struct Interrupt *)
                                ((i)->is_Node.ln_Succ)
#define LAST(i)     (((i)->is_Node.ln_Succ)->ln_Succ == 0)

struct ExecBase *EB;

char *IntrName[] =
{
    "TBE",      "DISKBLK", "SOFTINT", "PORTS",
    "COPER",    "VERTB",   "BLIT",    "AUD0",
    "AUD1",     "AUD2",    "AUD3",    "RBF",
    "DSKSYNC", "EXTER",    "INTEN",   "NMI"
};

char IntrPri[] = {1,1,1,2,3,3,3,4,4,4,4,5,5,6,6,7};

main()
{
    extern VOID *OpenLibrary();

    puts("Guru's Guide Interrupt Table Printer\n");

    EB = OpenLibrary("exec.library", 0);
    if (EB == NULL) exit(100);

    PrintAllIntr();
}

char *GetNodeName(node)
    REG struct Node *node;
{
    if (node == NULL) return "";

    if (node->ln_Name == NULL) return "(missing)";
    else return node->ln_Name;
}

int GetNodePri(node)
    struct Node *node;
{
    if (node == NULL) return 0;

    return node->ln_Pri;
}
```

```
PrintAllIntr()
{
    REG int i;
    REG struct IntVector *iv = &EB->IntVects[0];
    APTR sc = (APTR) EB->IntVects[3].iv_Code;

    puts("IV E INTR    P S   CODE     DATA   NPR NAME");
    puts("-- - ------- - - -------- -------- --- ----");

    for (i = 0; i < 16; i++, iv++)
    {
        printf("%2d %c %-7s %d ", i,
            (custom.intenar & (1 << i)) ? '*' : ' ',
            IntrName[i],
            IntrPri[i]);

        if (iv->iv_Code == sc)
        {
            PrintServers(iv->iv_Data);
        } else if (iv->iv_Code != 0)
        {
            printf("  %08lx %08lx     %s\n",
                iv->iv_Code,
                iv->iv_Data,
                GetNodeName(iv->iv_Node));
        } else putchar('\n');
    }
}


PrintServers(slist)
    struct List *slist;
{
    REG struct Interrupt *s;

    if (EMPTY(slist))
    {
        puts("*");
        return;
    }

    for (s = FIRST(slist); NEXTINT(s) != 0;
                           s = NEXTINT(s))
    {
        printf("* %08lx %08lx%4d %s\n",
                s->is_Code,
                s->is_Data,
                GetNodePri(s),
                GetNodeName(s));

        if (!LAST(s)) printf("                ");
    }
}
```

3.5 Servers

As you remember from the *Principles* chapter, servers provide a means of sharing certain types of interrupts. Exec provides built-in server support for five interrupts[5]: PORTS, COPER, VERTB, EXTER, and NMI.

When a server interrupt occurs, it is directed to an Exec *Server Handler*. This handler interacts with the AIC and dispatches each of the servers linked to the interrupt. Because Exec deals with the AIC, servers are generally simpler to write than their handler counterparts.

As described above, only two functions are needed to manage a server: **AddIntServer** and **RemIntServer**. **AddIntServer** is called when you are ready for your server to become active. It links your server into the chain associated with a particular interrupt. **RemIntServer** is called when you are finished with the interrupt.

Environment

Servers execute within a well defined system environment. Exec begins to establish this environment from the moment it is started by the processor.

Exec saves several of the CPU registers before dispatching servers. The standard scratch registers: **D0, D1, A0, A1** are preserved by Exec[6], so your code can operate without needing to save them. In addition **A5** and **A6** are often used by servers, so they are also preserved by the system.

Of course, if you decide to use other registers, their values must be preserved. This is normally done by pushing the values on the stack at entry, and popping the values off at exit.

Two registers are passed to you with special values. The **A1** register holds a pointer to your data segment as stored in the

---

[5] In theory, you can create your own interrupt server handler for other, non-server type interrupts, but other programs may not abide by your change (in other words, other programs will use the handler functions, not server functions).

[6] There is a bug that requires special attention if you are using the VERTB server and your priority is greater than nine. In such cases you must preserve the value of A0. This value is used by the graphics library server at priority ten.

is_Data field of your interrupt structure. The **A5** register holds a pointer to your code's entry point (the **is_Code** field).

Another element of the environment is the CPU mode. Your server will be executing in supervisor mode on the supervisor stack. Because of this, you should take extra care in writing your server. Here is a summary of things to avoid. See the *Principles* chapter for a more detailed discussion.

- Beware of task dependencies. Your server may have interrupted any task in just about any state. The task could be in the middle of allocating memory or performing I/O. If you reference variables within your own task, know how and when these variables are accessed. You may have interrupted your task when a particular variable was being held in a CPU register and its memory value is out of date!
- Don't use most of the Amiga system library functions, they may not work properly in interrupts, or they may cause the system to crash.
- Avoid causing a processor trap (68000 exception) as it would summon the Guru.
- Don't use much stack space. The supervisor stack is relatively small in size, and interrupt nesting may have used a lot of it already.
- Don't mess with the AIC or the CPU interrupt priority level. Exec handles the AIC for you.

Chaining

Servers are chained together in a priority order – the highest priority being first on the list. The server is inserted into the correct spot in the list with **AddIntServer**.

Each server is called in turn until the end of the list is reached, or until a particular server decides to terminate the chaining process. It is the responsibility of each server to set the processor's zero flag to indicate whether chaining should continue or be terminated. *The zero flag must be set or cleared intentionally!* Do not let it be set randomly by the execution of your server code. To continue the chaining process, exit with an instruction sequence similar to:

```
moveq    #0,d0    ; continue chaining
rts
```

To terminate chaining, use:

```
moveq    #1,d0    ; terminate chaining
rts
```

Remember that some instructions (like MOVE to an address register) do not affect the zero flag.

For some interrupts (like VERTB) all servers should continue the chain. For other interrupts, it will depend on the hardware involved.

Server Priorities

The priority of your server is established by the value in the **ln_Pri** byte of your interrupt structure at the moment **AddIntServer** is called. This priority value will determine the placement of your server relative to others in the same chain; hence, it will affect the time delay (latency) between when the interrupt occurred and when your code actually got executed.

Servers with higher priorities will be dispatched first. Servers with equal priorities will be dispatched in a FIFO[7] order. The first server linked with AddIntServer will be the first executed.

Most servers are given a priority of zero. This is a neutral priority indicating that the server really doesn't care when it's executed.

Priorities greater than zero are used when the server needs attention relatively soon after the interrupt occurs. You may put a server at level five because it needs to avoid the "crowd" at level zero, or you may put it at level 100 so it has a good chance of running first.

Some hardware may need software interaction within a few hundred microseconds after the interrupt occurs. For chains like PORTS and EXTER, it may make sense to use a higher priority for special hardware that interrupts more frequently and properly terminates the chain (thus reducing system overhead).

Lower priorities (less than zero) indicate that the server doesn't need quick attention or does not want to interfere with others. This is often the case for interrupt monitoring code or for examples (as you will soon see).

---

[7] First in, first out.

The general rule here is to be considerate about the value you select. Don't use a priority higher than what you really need.

Examples

Well, if all this theory and discussion are starting to wear you down, maybe it's time for a brief sojourn in C. Here are two of the simple working examples. The purpose of these programs is to count vertical blanking interrupts and print out the counter as it changes.

The first example makes use of a static interrupt data structure, the second uses a dynamically allocated structure (as defined in the structure section at the beginning of this chapter). Both are approximately the same degree of difficulty (simple).

The toughest part of these examples is compiling them. With the Manx, Aztec C compiler use the commands:

```
cc +L simple.c
ln simple.o -lc32
```

If you have some other compiler, adjust this as necessary. The +L flag indicates that you want long (32 bit) integers as the default. When interfacing to most Amiga library functions, this tends to make life easier. It lessens the chances of passing the wrong size integer to Amiga functions. Of course, you pay a small price for this convenience: the resulting program will be slightly larger. If your time is valuable, +L is the way to go. If you decide not to use +L, you may need to precede the **INTB_VERTB** constants with a **(long)** coersion; otherwise you may end up skewing the stack by a 16 bit word. Read more about this in your compiler's manual.

A few notes:

- The server code is in assembly. Short, simple, fast.
- The server code does not reference values through the compiler's data segment pointer, so we don't need to set it up and restore it.
- The Manx **Enable_Abort** flag is used to prevent a CTRL-C abort while the program is running. We wouldn't want to abort with the server still installed.

```
/*************************************************/
/***  Guru's Guide Example: Simple Interrupt    ***/
/*************************************************/

#include <exec/interrupts.h>
#include <hardware/intbits.h>

VOID    IntrProc(); /* forward to interrupt code */
long    IntrCount;  /* vertical blank counter    */

/* Initialized interrupt structure */
struct Interrupt VertBlank =
{
    NULL,                   /* node pred*/
    NULL,                   /* node succ*/
    -60,                    /* node pri */
    NT_INTERRUPT,           /* node type*/
    "simple.example",       /* node name*/
    (APTR)&IntrCount,       /* data ptr */
    &IntrProc               /* code ptr */
};


#asm    /*  Interrupt Server Code */
_IntrProc:
            addq.l  #1,(a1) ; increment IntrCount
            moveq   #0,d0   ; Z-flag - do next server
            rts
#endasm


main()  /*  Initialization, Main, and Clean-up */
{
    extern int Enable_Abort;

    Enable_Abort = 0;   /* prevent a CTRL-C */

    IntrCount = 0;

    AddIntServer(INTB_VERTB, &VertBlank);

    while (IntrCount < 200) printf("%d\n",IntrCount);

    RemIntServer(INTB_VERTB, &VertBlank);
}
```

Notice that this example uses initialized data for the interrupt structure, so it does not need code to initialize it.

A reminder: don't forget the '&' in front of the structure references (like I did when I first wrote this example). You'll blow off plenty of time trying to figure out what went wrong.

```
/************************************************/
/***  Guru's Guide Example: Simpler Interrupt   ***/
/************************************************/

#include <exec/interrupts.h>
#include <hardware/intbits.h>

VOID    IntrProc(); /* forward to interrupt code */
long    IntrCount;  /* vertical blank counter    */


#asm    /*  Interrupt Server Code */
_IntrProc:
            add.l   #1,(a1) ; increment IntrCount
            moveq   #0,d0   ; Z-flag - do next server
            rts
#endasm


main()  /*  Initialization, Main, and Clean-up */
{
    extern int Enable_Abort;
    extern struct Interrupt *MakeIntr();
    struct Interrupt *intr;

    Enable_Abort = 0;   /* prevent a CTRL-C */

    IntrCount = 0;

    intr = MakeIntr("simple.example", -60,
                        &IntrProc,&IntrCount);
    if (intr == NULL) exit(100);

    AddIntServer(INTB_VERTB, intr);

    while (IntrCount < 200) printf("%d\n", IntrCount);

    RemIntServer(INTB_VERTB, intr);

    FreeIntr(intr);
}
```

This example accomplishes the same result as the first but uses the **MakeIntr** and **FreeIntr** functions defined in the *Data Structure* section. To link this example, you will need to add those functions as an additional link input file:

```
ln simple.o intr.o -lc32
```

3.6 Task Signals

Frequently within interrupts, you will want to perform operations that are either too time consuming or are not permitted within supervisor mode. In such cases, you will need to forward the interrupt back to a task for further processing. On the Amiga the simplest way to do this is with *Task Signals* [8]. Even though this volume of the Guru's Guide does not discuss tasks in general, signals are believed to be relevant enough to be addressed here.

Signals are the backbone for intertask synchronization and communication on the Amiga. The heavily used message system relies on signals to notify tasks of message delivery.

Signals are task relative; that is, they only have meaning to the task in which they were defined. If you want to signal a task, it must tell you in advance what signal to use.

To a task, each signal is represented as a single bit in a long word mask. Thus, each task may have up to 32 independent signals. Of these, 16 are reserved for the system. The remaining 16 are available to the programmer.

You can think of signals as being very similar to interrupts. A task can wait for one or more signals to occur. If the task is busy the signals will remain pending until they have been given attention. Unlike interrupts, signals do not usually interrupt the immediate flow of control in the executing task[9].

Because of their simple structure, signals offer low overhead and high performance for interrupt processing.

---

[8] This isn't the only way however. Messages are often used to the same end, but they require a bit more work and care.

[9] Exec's Task Exceptions actually allow this. They allow signals to be task relative interrupts.

Functions

A task can receive many signals at the same time. It can either wait for the signal or periodically poll for it. The following Exec library functions support task signals:

Signal is used to send a signal to a task.

Wait tells a task to delay until one of many signals has occurred.

SetSignal lets a task query and alter the states of its current signals.

AllocSignal is used to allocate a signal from the pool available to a task.

FreeSignal is used to return an allocated signal to the task's pool.

For more information about these functions, refer to the *Amiga ROM Kernel Manual, Exec.*

Example

In this example, a signal is used to notify a task that the vertical blanking interrupt has occurred.

Two functions **MainInit** and **MainExit** were created to make the **main** function more clear.

Notice that **AllocSignal** and **FreeSignal** deal with the signal bit number. To be used with **Signal** and **Wait**, this bit number must converted to a flag (mask) bit. This is done with the left-shift operator (<<).

```
/*************************************************/
/***  Guru's Guide Example: Task Signal        ***/
/*************************************************/

#include <exec/exec.h>
#include <hardware/intbits.h>

struct Interrupt *Intr = NULL;
long ASignal = -1;

/* Structure used by interrupt */
struct SigInfo
{
    long counter;
    long signal;
    struct Task *task;
} IntrData;


/* Interrupt Processing Code */
VOID    IntrProc();
#asm
EXECBASE    equ 4
            public  _LVOSignal
_IntrProc:
            addq.l  #1,(a1)+    ; counter
            move.l  (a1)+,d0    ; signal
            move.l  (a1)+,a1    ; task
            move.l  EXECBASE,a6
            jsr     _LVOSignal(a6)
            moveq   #0,d0    ; Z-flag - do next server
            rts
#endasm

main()
{
    int count;

    MainInit();

    AddIntServer(INTB_VERTB, Intr);

    for (count = 0; count < 200; count++)
    {
        Wait(IntrData.signal);
        printf("%d %d\n", count, IntrData.counter);
    }

    RemIntServer(INTB_VERTB, Intr);

    MainExit();
}


MainInit()
{
    extern struct Interrupt *MakeIntr();
```

```
    extern long AllocSignal();
    extern struct Task *FindTask();
    extern int Enable_Abort;

    Enable_Abort = 0;    /* prevent a CTRL-C */

    Intr = MakeIntr("signal.example",-60,
                    &IntrProc,&IntrData);
    if (Intr == NULL) MainExit(201);

    ASignal = AllocSignal(-1);
    if (ASignal == -1) MainExit(202);

    IntrData.counter = 0;
    IntrData.signal = 1 << ASignal;
    IntrData.task = FindTask(NULL);
}


MainExit(error)
    int error;
{
    FreeIntr(Intr);

    if (ASignal != -1) FreeSignal(ASignal);

    exit(error);
}
```

There are two independent counters in this program. One counts the number of times the task has waited for a signal. The other counts the number of vertical blank interrupts. Notice that the two counters do not agree. The difference is due to the time needed to display the numbers. It takes several vertical blanking periods to do all of the processing associated with screen I/O.

Coprocessor Example

Here is another, more involved example. This program sets-up a coprocessor interrupt to occur when the video beam hits line 100 of the screen. Each time the interrupt occurs, a counter is decremented. When the counter reaches zero, a signal is sent and the program is terminated.

```
/*************************************************/
/***  Guru's Guide Example: Copper Interrupt    ***/
/*************************************************/

#include <exec/exec.h>
#include <hardware/custom.h>
#include <hardware/intbits.h>
#include <graphics/gfxmacros.h>
#include <graphics/copper.h>
#include <intuition/intuition.h>

APTR GfxBase = NULL;
APTR IntuitionBase = NULL;

struct Task *ATask = NULL;
struct Interrupt *Intr = NULL;
struct Screen *AScreen = NULL;
struct ViewPort *VPort = NULL;
struct UCopList *CoprList = NULL;
struct UCopList *SaveList = NULL;
long ASignal = -1;

long Count = 120;

/* Intuition Screen Specification */
struct NewScreen ScreenSpec =
{
    0,0,
    320,200,4,
    0,0,0,
    CUSTOMSCREEN,
    0,
    (UBYTE *) "Copper Example",
    0
};

/* Interrupt Processing Code */
VOID IntrProc()
{
    int_start();

    if (--Count <= 0) Signal(ATask,1 << ASignal);

    int_end();
}

main()
{
    MainInit();

    VPort = &AScreen->ViewPort;

    /* Save old user copper list */
    SaveList = VPort->UCopIns;

    /* Build a Copper List */
```

```
    CINIT(CoprList,100);
    CWAIT(CoprList,100, 0); /* Video Line 100 */
    CMOVE(CoprList,custom.intreq,INTF_SETCLR|INTF_COPER);
    CEND(CoprList);

    /* Insert new user copper list */
    VPort->UCopIns = CoprList;
    MakeScreen(AScreen);
    RethinkDisplay();

    /* Setup copper interrupt */
    AddIntServer(INTB_COPER, Intr);

    /* Something else to do... */
    while (Count > 0) printf("%d\n",Count);

    Wait(1 << ASignal); /* Sync-up */

    /* Clean up */
    RemIntServer(INTB_COPER, Intr);

    MainExit(0);
}


MainInit()
{
    extern APTR OpenLibrary();
    extern struct Screen *OpenScreen();
    extern struct Interrupt *MakeIntr();
    extern long AllocSignal();
    extern struct Task *FindTask();
    extern void *AllocMem();
    extern int Enable_Abort;

    Enable_Abort = 0;   /* prevent a CTRL-C */

    GfxBase = OpenLibrary("graphics.library", 30);
    if (GfxBase == NULL) MainExit(201);

    IntuitionBase = OpenLibrary("intuition.library", 30);
    if (GfxBase == NULL) MainExit(202);

    AScreen = OpenScreen(&ScreenSpec);
    if (AScreen == NULL) MainExit(210);

    CoprList = AllocMem(sizeof(*CoprList), MEMF_CLEAR);
    if (CoprList == NULL) MainExit(220);

    ASignal = AllocSignal(-1);
    if (ASignal == -1) MainExit(230);

    Intr = MakeIntr("copper.example",0,&IntrProc,0);
    if (Intr == NULL) MainExit(240);

    ATask = FindTask(NULL);
}
```

```
MainExit(error)
    int error;
{
    if (AScreen != NULL)
    {
        VPort->UCopIns = SaveList; /* restore it */
        MakeScreen(AScreen);
        RethinkDisplay();
        CloseScreen(AScreen);
    }

    if (CoprList != NULL)
    {
        /* Free intermediate copper list */
        FreeCopList(CoprList->FirstCopList);

        FreeMem(CoprList,sizeof(*CoprList));
    }

    FreeIntr(Intr);

    if (ASignal != -1) FreeSignal(ASignal);

    if (IntuitionBase != NULL)
        CloseLibrary(IntuitionBase);

    if (GfxBase != NULL)
        CloseLibrary(GfxBase);

    exit(error);
}
```

3.7 Controller

The Amiga Interrupt Controller (AIC) was described in detail in the *Principles* chapter. As you remember, the AIC supplies four registers (two for read, two for write) to support four basic operations: enable, disable, set request, clear request. To summarize the *Principles* chapter, the registers are:

INTENA to enable and disable (write only)

INTENAR to query the enable states (read only)

INTREQ to request and clear (write only)

INTREQR to query the request states (read only)

Each of these registers is a word in size (16 bits).

To use the AIC, your code must read and write these registers. The system does not provide functions to access these registers, so your code will change them directly. When doing this, *take care never to read one of the write only registers.* This would cause a random value to be written into the register.

Include Files

Most Amiga system-programming languages provide include files with structures, defines and equates for accessing the AIC registers. These include files are called **custom** and **intbits** and can usually be found in the **include/hardware** directory. If you are coding in C the include file specifications would look like:

```
#include <hardware/custom.h>
#include <hardware/intbits.h>
```

The first file gives you access to the AIC registers. It defines a C structure called **Custom** which specifies the hardware registers as structure field names. This makes it relatively easy to access the registers from C. You can now create a C variable to reference this structure:

```
struct Custom *custom;
```

This would let you reference the interrupt registers indirectly:

```
custom->intena
custom->intenar
custom->intreq
custom->intreqr
```

Or, you could define a C macro to access the registers directly:

```
#define custom (*((struct Custom *) 0xdff000))

custom.intena
custom.intenar
custom.intreq
custom.intreqr
```

The second include file (*intbits.h*) defines the names for each of the interrupts. It contains two ways to reference an interrupt: as a bit number or as a mask. The bit numbers begin with the prefix "**INTB_** ". Here is a list of the interrupt bits numbers and their names:

```
15 INTB_SETCLR    -   set and clear the bit mask
14 INTB_INTEN     -   master interrupt (enable only )
13 INTB_EXTER     -   CIA-B + external (high pri)
12 INTB_DSKSYNC   -   disk word synchronized
11 INTB_RBF       -   serial, receive buffer full
10 INTB_AUD3      -   audio channel 3 ready
 9 INTB_AUD2      -   audio channel 2 ready
 8 INTB_AUD1      -   audio channel 1 ready
 7 INTB_AUD0      -   audio channel 0 ready
 6 INTB_BLIT      -   blitter finished
 5 INTB_VERTB     -   start of vertical blank
 4 INTB_COPER     -   coprocessor
 3 INTB_PORTS     -   CIA-A + external (low pri)
 2 INTB_SOFTINT   -   software interrupt
 1 INTB_DSKBLK    -   disk block done
 0 INTB_TBE       -   serial, transmitter buffer empty
```

The interrupt masks have the same names as those above, but with the prefix **INTF** (F for Flag) rather than **INTB**. A mask is simply a 16 bit word with the interrupt bit already set.

Interrupt bit numbers (**INTB**) are mostly used as arguments to the system interrupt functions, whereas the masks (**INTF**) are used for operations performed directly on the interrupt registers.

Set/Clear

A special technique is used to modify bits in the INTENA and INTREQ registers. The most significant bit (#15) in these registers is called the **SETCLR** bit. It is used in conjunction with a 14 bit mask to change individual bits in the register.

When the **SETCLR** bit is one, all one bits in the mask will be set. When **SETCLR** is zero, all one bits in the mask will be cleared. With this mechanism, several bits can be changed at the same time without affecting the other bits.

Figure 3.2 illustrates how writing a pattern of 8020 hex will enable the vertical blanking interrupt without affecting the other interrupts. In this case the mask is 0020 hex, so only the vertical blanking interrupt is to be affected. Its new value is derived from bit 15, which is set to one (8000 hex).

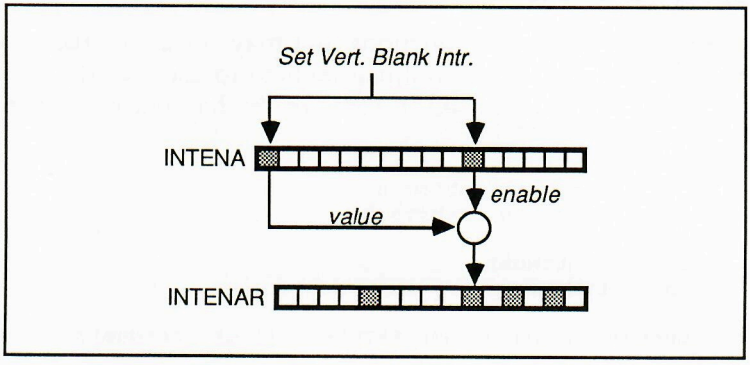

Fig 3.2 – Enabling with Set/Clear

Using this method of changing bits, you can enable, disable, request, and clear a single interrupt. You do not need to know the status of the other interrupts before doing so. This makes interrupt enabling and disabling almost trivial, and saves the system considerable overhead.

For example, to enable the **AUD0** interrupt from C:

```
custom.intena = INTF_SETCLR | INTF_AUD0;
```

or in assembly code:

```
move.w   #(INTF_SETCLR|INTF_AUD0),_intena
```

To disable the same interrupt:

```
custom.intena = INTF_AUD0;
```

or in assembly code[10]:

```
move.w   #INTF_AUD0,_intena
```

Sampling the interrupt request would be:

```
if (custom.intenar & INTF_AUD0) {...}
```

To clear the interrupt after processing:

[10] Never use a BSET or BCLR instruction as it would perform a read operation on a write only register. This would write garbage to the register.

```
custom.intreq = INTF_AUD0;
```

Or, to artificially cause the interrupt:

```
custom.intreq = INTF_SETCLR | INTF_AUD0;
```

Support Functions

Here are a few "support" functions that may come in handy for using the AIC (and they also illustrate how to use the AIC). These functions will be used in some of the handler examples that follow.

```
#include <hardware/custom.h>
#include <hardware/intbits.h>

EnableIntr(intrNum)
    int intrNum;
{
    custom.intena = INTF_SETCLR | (1 << intrNum);
}


DisableIntr(intrNum)
    int intrNum;
{
    custom.intena = (1 << intrNum);
}


RequestIntr(intrNum)
    int intrNum;
{
    custom.intreq = INTF_SETCLR | (1 << intrNum);
}


ClearIntr(intrNum)
    int intrNum;
{
    custom.intreq = (1 << intrNum);
}
```

Using these functions you could enable the Audio 0 channel with:

```
EnableIntr(INTB_AUD0);
```

Master Enable

Bit 14 (INTEN) of the INTENA register is the master enable for interrupts. When set to zero this bit disables all interrupts. When set to one this bit enables all interrupts that have their enable bits set.

This is the one enable bit that programs are not permitted to change directly. This bit is used by the system software to disable interrupts during critical sections. If your code modified the master enable, it might inadvertently turn interrupts back on at the wrong time. This could create serious, intermittent reliability problems for the system software.

Two system functions[11] are provided to solve this problem. The **Disable** function shuts off all interrupts; **Enable** turns them back on. These functions can be nested. For example, if **Disable** is called three times, then **Enable** must be called three times before interrupts actually become enabled again. This prevents interrupts from becoming enabled within a critical section.

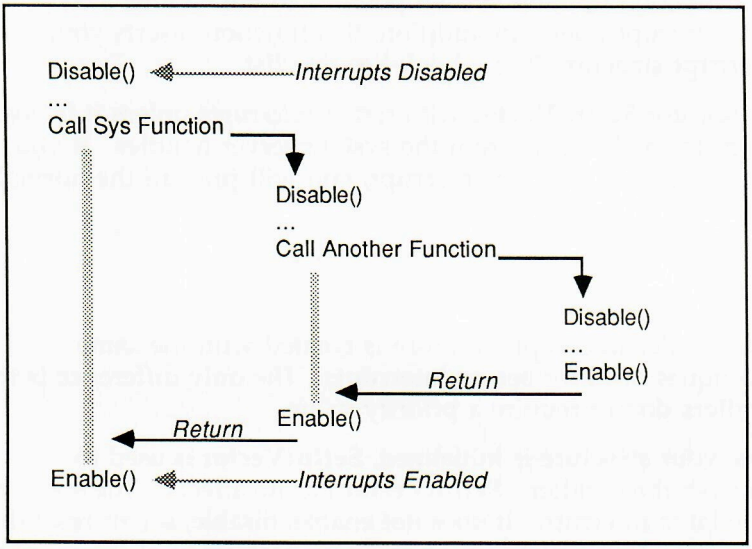

Fig 3.3 – Nested Disables

[11] Macros are also available for performance.

3.8 Handlers

Handlers provide a direct connection to interrupts that cannot be shared. Handlers usually interact with the AIC and thus require more attention to details than a typical server. Handlers are used to process interrupts for the serial port (RBF and TBE), disk (sync and block), audio channels, and blitter.

As described earlier in this chapter, only one Exec function is responsible for managing handlers: **SetIntVector**. This function replaces the Exec interrupt vector with values that you supply.

This function updates system pointers to the handler's code, data, and interrupt node. In addition, this function inserts your interrupt structure into a book-keeping list.

Do not use **SetIntVector** with *server interrupts* unless it is your intent to steal control from the system server handler. If you take control of a server interrupt, you will prevent the normal execution of its servers.

Setup

The handler interrupt structure is created with the same techniques used for server interrupts. The only difference is that handlers do not require a priority value.

Once your structure is initialized, **SetIntVector** is used to establish the handler. **SetIntVector** has no affect on the state of the related interrupt. It does not enable, disable, set, or reset the interrupt. These operations should be performed at the proper times by your code.

To safely setup handlers, you might want to create a special support function called **AddHandler**:

```
#include <exec/interrupts.h>

extern struct Interrupt *SetIntVector();

struct Interrupt *AddHandler(intrNum,intr)
    int intrNum;
    struct Interrupt *intr;
{
    DisableIntr(intrNum);
    ClearIntr(intrNum);
    SetIntVector(intrNum,intr); /* ignore result */
}
```

This code disables the interrupt and clears a possible pending interrupt before setting the vector. This is done for safety. The interrupt could be enabled and your code may not be ready to handle the interrupt quite yet. When ready, enable the interrupt to make it active.

In this code, the result of **SetIntVector** is ignored. You could add a couple lines here to check the result and print a message if you took the interrupt away from some other program.

Note: The system copies the code and data fields of a handler's interrupt structure into its own table. Because of this, modifying your code and data pointers after calling this function will have no direct affect on the handler execution.

Clean-Up

When your code has finished, it should remove its handler. Calling **SetIntVector** with a zero for the structure pointer will remove you from the book-keeping list and delete the pointers to your code and data.

The **RemHandler** support function listed below will help:

```
RemHandler(intrNum,intr)
    int intrNum;
    register struct Interrupt *intr;
{
    register struct Interrupt *retIntr;

    if (intr == NULL) return NULL;

    /*  CRITICAL SECTION:
    **  Mutually exclude all tasks and interrupts.
    **  Remove the handler if it is really ours
    **  (it could have been replaced already!)
    **  Disable the interrupt if it is ours.
    */
    Disable();
    retIntr = SetIntVector(intrNum,NULL);
    if (retIntr != intr) SetIntVector(intrNum,retIntr);
    else DisableIntr(intrNum);
    Enable();
}
```

Notice the critical section. This deals with the possibility that your code lost possession of the interrupt to some other program. In that case, you wouldn't want to remove the active handler because it doesn't belong to you.

Environment

Like servers, handlers also execute within a well defined system environment. This environment is quite similar to that used by servers.

The CPU registers **D0**, **D1**, **A0**, **A1**, **A5**, and **A6** are saved and available to you. All other registers must be preserved. This is normally done by pushing their values on the stack.

Five registers are passed to you with special values. As with servers, the **A1** register holds a pointer to your data segment, and the **A5** register holds a pointer to your code. In addition, **A0** points to the base of the custom chips (handy for accessing the hardware, including AIC), **A6** points to the Exec library base, and **D1** indicates the currently active interrupts (it is equal to INTREQ anded with INTENA).

The rest of the environment is identical to that of servers. Read that section for more information.

Audio Channel Example

This example plays a series of notes through one of the Amiga audio channels. An audio interrupt (AUD0) is used to track the waveform and make volume changes for an ADSR (attack, decay, sustain, release) envelope.

A word of caution is necessary: on the Amiga, sounds are usually created through the Audio device. This device manages the allocation of all Amiga audio channels. *If you plan to create commercial quality programs, it is advised that you read the documentation for the Audio device.* This example program steals the AUD0 interrupt vector. This action defeats the purpose of the Audio device, but we wanted to keep the example as short as possible. *The example is provided here for educational and experimental purposes; it should not be included as part of a product for either commercial or public domain distribution.*

```
/*************************************************/
/***  Guru's Guide Example: Audio Interrupt     ***/
/*************************************************/

#include <exec/exec.h>
#include <hardware/custom.h>
#include <hardware/dmabits.h>
#include <hardware/intbits.h>

#define SIZE 256    /* wave size */

struct Interrupt *Intr = NULL;
struct Task *ATask = NULL;
long ASignal = -1;

char *WaveForm = NULL;

short RepCount; /* wave rep counter */
short Part;     /* ADSR envelope count */
short Volume;

/* Attack, decay, sustain, release tables: */
char ADSR[] = { 3, 3, 5,20 }; /* Reps */
char Ramp[] = {20,-5, 0,-2 }; /* Incr */

/* Periods for several "notes" */
short Period[] = { 100,200,300,400,3000,0 };


/* Interrupt Processing Code */
VOID IntrProc()
{
    int_start();

    /* Next ADSR part? */
    if (--RepCount <= 0)
    {
        ++Part;
        RepCount = ADSR[Part];
    }

    /* Finished with note? */
    if (Part > 3)
    {
        Volume = 0;
        DisableIntr(INTB_AUD0);
        Signal(ATask,1 << ASignal);
    }
    else /* Change volume & clip */
    {
        Volume += Ramp[Part];
        if (Volume < 0) Volume = 0;
        else if (Volume > 63) Volume = 63;
    }

    custom.aud[0].ac_vol = Volume;
    ClearIntr(INTB_AUD0);
```

```
    int_end();
}


main()
{
    struct AudChannel *ac;
    int n;

    MainInit();

    /* Generate unfiltered noise: */
    for (n = 0; n < SIZE; n++)
        WaveForm[n] = (char)rand();

    AddHandler(INTB_AUD0, Intr);

    /* Turn on audio DMA */
    custom.dmacon = DMAF_SETCLR | DMAF_AUD0 |
                                  DMAF_MASTER;

    ac = &custom.aud[0];
    ClearIntr(INTB_AUD0);

    for (n = 0; Period[n] > 0; n++)
    {
        Volume = 0;
        Part = 0;
        RepCount = ADSR[0];
        ac->ac_per = Period[n];
        ac->ac_ptr = (UWORD *)WaveForm;
        ac->ac_len = SIZE/2;
        ac->ac_vol = Volume;

        EnableIntr(INTB_AUD0);

        Wait(1 << ASignal);
    }

    /* Turn off audio DMA */
    custom.dmacon = DMAF_AUD0;

    RemHandler(INTB_AUD0, Intr);

    MainExit(0);
}


MainInit()
{
    extern struct Interrupt *MakeIntr();
    extern long AllocSignal();
    extern struct Task *FindTask();
    extern void *AllocMem();
    extern int Enable_Abort;

    Enable_Abort = 0;   /* prevent a CTRL-C */
```

```
    Intr = MakeIntr("audio.example",0,&IntrProc,0);
    if (Intr == NULL) MainExit(201);

    /* Audio DMA needs waveform in chip memory */
    WaveForm = AllocMem(SIZE, MEMF_CHIP | MEMF_CLEAR);
    if (WaveForm == NULL) MainExit(202);

    ASignal = AllocSignal(-1);
    if (ASignal == -1) MainExit(203);

    ATask = FindTask(NULL);
}


MainExit(error)
    int error;
{
    FreeIntr(Intr);

    if (WaveForm != NULL) FreeMem(WaveForm, SIZE);

    if (ASignal != -1) FreeSignal(ASignal);

    exit(error);
}
```

Serial Port Example

This example illustrates the use of a handler interrupt for reading characters from the Amiga RS232 serial port. Each time a character arrives at the serial port, an interrupt occurs. The handler for this interrupt, saves the character, clears the interrupt, and signals the main task. The handler is executing at level five, so its code is written in assembly.

If a character does not arrive within an allotted period of time, a timeout signal occurs and a message is displayed. The timeout signal is generated from a counter connected as a server to the vertical blanking interrupt.

This example sets the serial port up to receive at 9600 baud in an unbuffered fashion (only one character at a time). You will experience overruns if you send characters too fast. To prevent this it would be wise to add a buffer for storing many characters from within the interrupt handler. A circular buffer works well for this. We did not include a buffer – it would have made the example too long.

Important note: this example is for educational and experimental purposes. If you want to create code for a product, you should use the serial device driver and the timer device, not this code.

```
/************************************************/
/***  Guru's Guide Example: Serial Interrupts   ***/
/************************************************/

#include <exec/exec.h>
#include <hardware/intbits.h>
#include <hardware/custom.h>
#include <stdio.h>

#define TIMEOUT 240     /* 4 seconds */
#define BAUD_RATE 372   /* 9600 baud */

char IntrName[] = "serial.example";

struct  SigInfo
{
    short   buffer;     /* char buffer */
    short   intrmask;   /* intr to reset */
    long    signal;     /* signal mask */
    struct  Task *task; /* task to signal */
    long    sig;        /* sig bit to free */
    struct  Interrupt *intr;
} Serial, TimeOut;


/* Interrupt Processing Code */
VOID    SerialCode();
VOID    TimeOutCode();
#asm
TIMEOUT     equ 240
EXECBASE    equ 4
intreq      EQU $9C ; interrupt request
serdatr     EQU $18 ; serial data read

            public  _LVOSignal

_TimeOutCode:
            subq.w  #1,(a1)         ; decr timeout
            move.w  (a1),d0
            bge.s   exit            ; expired?
            move.w  #TIMEOUT,(a1)   ; reset timeout
            addq.l  #4,a1           ; adjust pointer
            bra.s   signal

_SerialCode:
            move.w  serdatr(a0),(a1)+ ; get the char
            move.w  (a1)+,intreq(a0); clear the request

signal:
            move.l  (a1)+,d0        ; the signal
            move.l  (a1)+,a1        ; the task
            move.l  EXECBASE,a6
            jsr     _LVOSignal(a6)
exit:
            moveq   #0,d0     ; Z-flag - do next server
            rts
#endasm
```

```
main()
{
    extern long Wait();
    extern int Enable_Abort;
    long waitsigs;
    char c;

    Enable_Abort = 0;    /* prevent a CTRL-C */

    /* Required for proper clean-up: */
    Serial.sig = TimeOut.sig = -1;

    MakeSigIntr(IntrName,INTB_RBF,0,&SerialCode,&Serial);
    MakeSigIntr(IntrName,INTB_VERTB,60,&TimeOutCode,
                                              &TimeOut);

    /* Set serial baud rate to 9600 */
    custom.serper = BAUD_RATE;

    AddHandler(INTB_RBF,Serial.intr);
    EnableIntr(INTB_RBF);

    TimeOut.buffer = TIMEOUT;
    AddIntServer(INTB_VERTB,TimeOut.intr);

    while (TRUE)
    {
        waitsigs = Wait(Serial.signal | TimeOut.signal);

        if (waitsigs & Serial.signal)
        {
            c = (char)Serial.buffer;
            if (c == 3) break;  /* CTRL-C to stop */
            if (c == 13) c = 10;/* change CR to LF */
            putchar(c);
            fflush(stdout); /* force it out */

            /* Reset timeout */
            TimeOut.buffer = TIMEOUT;
        }

        if (waitsigs & TimeOut.signal)
            puts("\nTIMEOUT!");
    }

    DisableIntr(INTB_RBF);
    MainExit(0);
}


MakeSigIntr(iname,ibit,pri,iproc,isig)
    char *iname;
    int ibit;
    VOID (*iproc)();
    register struct SigInfo *isig;
{
    extern  struct Task *FindTask();
```

```
    extern struct Interrupt *MakeIntr();
    struct Interrupt *intr;

    isig->sig = AllocSignal(-1);
    if (isig->sig == -1) MainExit(100);

    intr = MakeIntr(iname,pri,iproc,isig);
    if (intr == NULL) MainExit(101);

    isig->buffer = 0;
    isig->intrmask = 1 << ibit;
    isig->signal = 1 << isig->sig;
    isig->task = FindTask(NULL);
    isig->intr = intr;
}


MainExit(error)
    int error;
{
    puts("Terminate!");

    if (Serial.intr != NULL)
    {
        RemHandler(INTB_RBF, Serial.intr);
        FreeIntr(Serial.intr);
    }

    if (TimeOut.intr != NULL)
    {
        RemIntServer(INTB_VERTB, TimeOut.intr);
        FreeIntr(TimeOut.intr);
    }

    if (Serial.sig != -1) FreeSignal(Serial.sig);

    if (TimeOut.sig != -1) FreeSignal(TimeOut.sig);

    exit(error);
}
```

3.9 CIA Interrupts

Amiga hosts two complex interface adaptors (CIA) containing port I/O and timers. Each CIA includes two interval timers, a time of day clock with alarm, two parallel ports, and a serial port[12].

Each of these CIA devices has the ability to generate its own interrupts which are passed to the AIC. One CIA (called CIA-A) generates interrupts at a low priority (CPU level 2), the other (CIA-B) at a high priority (level 6).

Each CIA has its own interrupt control register (ICR). When written to, this register permits separate enabling (and disabling) of CIA interrupts. When read, the ICR indicates what interrupts are pending. Unfortunately, reading the ICR also clears all pending interrupts. This means that interrupt processing routines cannot independently read the ICR. For example, if both serial and timer interrupts were pending, and the timer interrupt routine read the ICR first, it would clear the serial interrupt in addition to its own. To solve this problem, a single central interrupt handler was created for each CIA device, and the CIA resources came into being.

On the Amiga, *Resources* are system modules which provide protocols for shared hardware in a multitasking environment. In structure they are somewhat similar to libraries, but they are simpler to implement and use. There are two resources associated with the CIAs: **ciaa.resource** and **ciab.resource**.

The CIA resources include a function dispatch table and a data segment. The data segment maintains enable and active masks for the ICR. All code operating on the ICR must update these masks.

The CIAs are heavily used on the Amiga. Many of the internal and external control signals come from these chips. For the most part, the devices on both CIAs have been allocated for use by the system. However, your program can use these devices if the system doesn't happen to be using them.

---

[12] Further information about the CIAs can be found in the *Amiga Hardware Reference Manual.*

For example, CIA-B allocates a timer for a serial communication port baud rate generator. This timer can be used by your code[13] if it is able to successfully perform the **AddICRVector** operation.

Here's how the CIA timers are normally allocated:

CIA-A

Port A	Miscellaneous control signals. Mainly floppy disk and gameport controls.
Port B	Amiga parallel port.
Serial Port	Amiga keyboard communication.
Timer A	Baud rate generator for the keyboard serial port. Use this and lose your keyboard.
Timer B	System tick timer. Used for the timer device and Exec task time-sharing.
TOD	Not used.

CIA-B

Port A	Serial port and printer control signals.
Port B	Floppy disk control signals (Flag interrupt is used for disk resource also).
Serial Port	Not used (Was reserved for Commodore serial communications – not the Amiga serial port, but some other type of I/O.)
Timer A	Not used. (Baud rate control for Commodore serial communications, if it ever happens)
Timer B	Not used.
TOD	Graphics library timing for a horizontal sync counter in some types of video operations.

Functions

Four functions are provided by each CIA resource. Just as with libraries, resources must be opened prior to their use. The Exec

[13] Don't forget that the timer device provides plenty of handy timing functions without the restrictions of using the CIA timers directly.

OpenResource function operates in an identical fashion to **OpenLibrary**. It accepts the name of the resource and a version number.

Once the resource has been opened, four functions can be called:

AddICRVector links an interrupt structure to a particular CIA interrupt and enables it.

RemICRVector unlinks an interrupt structure from a particular CIA interrupt and disables it.

AbleICR is used to separately enable and disable ICR interrupts (similar to INTENA).

SetICR provides a means of setting and clearing ICR interrupts (like INTREQ).

The following pages contain function descriptions for these.

AddICRVector

Summary

AddICRVector – link a new ICR handler

Invocation:

```
AddICRVector(cia,  ICRBit,  IStruct)
```

Accepts:

```
APTR  cia  (a6)
int  ICRBit   (D0)
struct  Interrupt  *IStruct   (A1)
```

Arguments

cia is the resource base pointer returned from the **OpenResource** function.

ICRBit is an integer used to indicate the ICR interrupt to link to. It must range from zero to four. The include files *cia.h* and *cia.i* in the *hardware* directory contain predefined constants for use with this function. They are of the form **CIAICRB**_*NAME*.

IStruct is a 32 bit pointer to an interrupt data structure. The structure must be properly initialized prior to calling this function. The interrupt structure is identical to that used for handlers and servers, but it does not require a priority.

Return

If this interrupt is not already in use, this function will return a zero. Otherwise, this function will fail and return a pointer to the current interrupt structure in control.

Description

This function links an interrupt structure to a particular CIA ICR interrupt, and the interrupt is enabled.

This function will not steal an ICR vector if it is already in use. Your code should not assume that it got the vector, it should check the return value of this function.

This function does not protect against incorrect parameters.

The structure must be unlinked with **RemICRVector** prior to exiting.

RemICRVector

Summary

RemICRVector – unlink an ICR handler

Invocation:

```
RemICRVector(cia,  ICRBit,  IStruct)
```

Accepts:

```
APTR  cia  (a6)
int  ICRBit   (D0)
struct  Interrupt  *IStruct   (A1)
```

Arguments

cia is the resource base pointer returned from the **OpenResource** function.

ICRBit is an integer used to indicate the ICR interrupt to unlink. It must range from zero to four. The include files *cia.h* and *cia.i* in the *hardware* directory contain predefined constants for use with this function. They are of the form **CIAICRB**_*NAME*.

IStruct is a 32 bit pointer to an interrupt data structure.

Description

This function unlinks an interrupt structure previously linked with **AddICRVector**. It will also disable the appropriate ICR interrupt.

This function does not protect against incorrect parameters. The structure must have been previously linked with **AddICRVector**.

AbleICR

Summary

AbleICR – enable/disable an ICR interrupt

Invocation:

```
AbleICR(cia,   ICRMask)
```

Accepts:

```
APTR  cia   (a6)
int   ICRMask    (D0)
```

Arguments

cia is the resource base pointer returned from the **OpenResource** function.

ICRMask is a byte containing the bits to be affected and a set/clear flag. If bit seven is set, then interrupts associated with each "on" bit in the mask are enabled. If bit seven is clear, then the associated interrupts are disabled.

Return

This function returns the previous ICR enable mask.

Description

This function enables or disables one or more ICR interrupts. It operates in a manner similar to the INTENA register, using a set/clear flag (CIAICRF_SETCLR) to indicate what interrupts to affect.

SetICR

Summary

SetICR – request/clear an ICR interrupt

Invocation:

```
SetICR(cia,  ICRMask)
```

Accepts:

```
APTR  cia   (a6)
int   ICRMask    (D0)
```

Arguments

cia is the resource base pointer returned from the **OpenResource** function.

ICRMask is a byte containing the bits to be affected and a set/clear flag. If bit seven is set, then interrupts associated with each "on" bit in the mask are requested. If bit seven is clear, then the associated interrupts are cleared.

Return

This function returns the previous ICR request mask.

Description

This function requests or clears one or more ICR interrupts. It operates in a manner similar to the INTENA register, using a set/clear flag (CIAICRF_SETCLR) to indicate what interrupts to affect.

ICR Bits

The ICR bit names are defined in the **hardware/cia** include files as:

```
0 CIAICRB_TA       - interval timer A
1 CIAICRB_TB       - interval timer B
2 CIAICRB_ALRM     - time of day alarm
3 CIAICRB_SP       - serial port
4 CIAICRB_FLG      - handshake flag
7 CIAICRB_IR       - intr request pending
7 CIAICRB_SETCLR   - set/clear the mask
```

The equivalent mask names begin with **CIAICRF_**.

CIA Example

This example uses Timer-A on CIA-B (high-priority) for a simple timing pulse. Notice the use of **AbleICR** and **SetICR**.

With this code running at CPU level six, it would probably be a good idea to recode the interrupt processing function in assembly

code rather than C. You really don't want the processor hanging out at level six for very long.

```
/*****************************************************/
/***  Guru's Guide Example: CIA Interrupt        ***/
/*****************************************************/

#include <exec/exec.h>
#include <hardware/intbits.h>
#include <hardware/cia.h>

struct Task *ATask = NULL;
struct Interrupt *Intr = NULL;
APTR Cia = NULL;
long ASignal = -1;

long Counter = 10;


/* Interrupt Processing Code */
VOID IntrProc()
{
    int_start();

    /* Clear timer-A interrupt */
    SetICR(Cia, CIAICRF_TA);

    if (--Counter <= 0)
    {
        /* Disable timer interrupt */
        AbleICR(Cia, CIAICRF_TA);

        Signal(ATask,1 << ASignal);
    }

    int_end();
}


main()
{
    MainInit();

    if (AddICRVector(Cia,CIAICRB_TA,Intr) != 0)
    {
        puts("CIA-B Timer-A in use.");
        MainExit(300);
    }

    /* At this point, Timer intr is linked  */
    /* and enabled, so it may have happened */
    /* already!  That's why we need this:   */
    ciab.ciacra &= ~CIACRAF_START;  /* stop timer */
    SetICR(Cia,CIAICRF_TA);     /* clear intr */
    SetSignal(0, 1 << ASignal); /* clear signal */

    /* Set timer counter latch */
    ciab.ciatalo = 0;
```

```
    ciab.ciatahi = 0xff;

    /* Start the timer */
    ciab.ciacra |= CIACRAF_START;

    while (Counter > 0)
        printf("%d %2x%02x\n", Counter, ciab.ciatahi,
                                        ciab.ciatalo);

    Wait(1 << ASignal); /* Sync-up */

    ciab.ciacra &= ~CIACRAF_START;  /* stop timer */

    RemICRVector(Cia,CIAICRB_TA,Intr);

    MainExit(0);
}


MainInit()
{
    extern APTR OpenResource();
    extern struct Interrupt *MakeIntr();
    extern struct Task *FindTask();
    extern int Enable_Abort;

    Enable_Abort = 0;   /* prevent a CTRL-C */

    Cia = OpenResource("ciab.resource", 0);
    if (Cia == NULL) MainExit(201);

    Intr = MakeIntr("cia.example",0,&IntrProc,0);
    if (Intr == NULL) MainExit(202);

    ASignal = AllocSignal(-1);
    if (ASignal == -1) MainExit(203);

    ATask = FindTask(NULL);
}


MainExit(error)
    int error;
{
    FreeIntr(Intr);

    if (ASignal != -1) FreeSignal(ASignal);

    exit(error);
}
```

3.10 Software Interrupts

As mentioned in the *Principles* chapter, software interrupts are used to off-load processing from higher priority interrupts or to serve as microtasks for important types of events.

Software interrupts operate in an environment that is identical to regular hardware interrupts and they are actually simpler to setup and use.

Software interrupts can be made to occur by calling the **Cause** function[14], or as an indirect result of sending an Exec message (**PutMsg**).

Priorities

For a software interrupt the node priority determines onto what queue it will be linked. Currently the priorities available for software interrupts are: +32, +16, 0, -16, -32. They should *never* be allowed to range outside +32 to -32, as this could crash the system. Values in-between those five listed will be rounded down (e.g. 31 would become 16, but -31 would become -32).

When queued to execute, a software interrupt structure has its node type changed automatically to **NT_SOFTINT**. This is done to prevent it from being linked more than once. When the interrupt has been removed from its queue immediately before it is executed, its type field is restored to **NT_INTERRUPT**.

SWI Example

This example shows how simple it is to setup and use a software interrupt.

[14] Described earlier in the System Functions section of this chapter.

```
/************************************************/
/***  Guru's Guide Example: Software Interrupt  ***/
/************************************************/

#include <exec/interrupts.h>

long Counter = 0;


/* Interrupt Processing Code */
IntrProc()
{
    int_start();

    Counter++;

    int_end();
}


main()
{
    extern struct Interrupt *MakeIntr();
    extern int Enable_Abort;
    struct Interrupt *intr;

    Enable_Abort = 0;    /* prevent a CTRL-C */

    intr = MakeIntr("softint.example",0,&IntrProc,0);
    if (intr == NULL) exit(100);

    while (Counter < 10)
    {
        Cause(intr);
        printf("%d\n", Counter);
    }

    FreeIntr(intr);
}
```

Message Example

Exec messages can be used to trigger software interrupts. To do this, the message port flags field must indicate a SWI action (**PA_SOFTINT**) instead of a normal signal action (**PA_SIGNAL**). When a message is posted to the port, the SWI will occur.

In the example that follows, a message causes a software interrupt (SWI). The SWI gets the message and sends it back as a reply. The reply message awakes the waiting task which then terminates.

```
/************************************************/
/***  Guru's Guide Example: Message Soft-Intr   ***/
/************************************************/

#include <exec/exec.h>

struct MsgPort *IPort = NULL;
struct MsgPort *RPort = NULL;
struct Interrupt *Intr = NULL;
long ASignal = -1;


/* Interrupt Processing Code */
IntrProc()
{
    extern struct Message *GetMsg();
    struct Message *msg;

    int_start();

    msg = GetMsg(IPort);
    if (msg != NULL) ReplyMsg(msg);

    int_end();
}


main()
{
    struct Message msg;

    MainInit();

    msg.mn_Node.ln_Type = NT_MESSAGE;
    msg.mn_Node.ln_Name = "swi.message";
    msg.mn_ReplyPort = RPort;

    puts("Causing interrupt...");
    PutMsg(IPort, &msg);

    puts("Awaiting reply...");
    WaitPort(RPort);

    puts("Got reply...");
    GetMsg(RPort);

    MainExit();
}


MainInit()
{
    extern struct Interrupt *MakeIntr();
    extern struct MsgPort *CreatePort();
    extern struct Task *FindTask();
    extern int Enable_Abort;

    Enable_Abort = 0;   /* prevent a CTRL-C */
```

```
    Intr = MakeIntr("softint.example",-16,&IntrProc,0);
    if (Intr == NULL) MainExit(201);

    ASignal = AllocSignal(-1);
    if (ASignal == -1) MainExit(202);

    IPort = CreatePort("swi.port", 0);
    if (IPort == NULL) MainExit(203);

    RPort = CreatePort("swi.reply.port", 0);
    if (RPort == NULL) MainExit(204);

    IPort->mp_SoftInt = (struct Task *) Intr;
    IPort->mp_Flags = PA_SOFTINT;

    RPort->mp_SigBit = ASignal;
    RPort->mp_SigTask = FindTask(NULL);
    RPort->mp_Flags = PA_SIGNAL;
}


MainExit(error)
    int error;
{
    FreeIntr(Intr);

    if (IPort != NULL) DeletePort(IPort);

    if (RPort != NULL) DeletePort(RPort);

    if (ASignal != -1) FreeSignal(ASignal);

    exit(error);
}
```

Index

68000 23, 25, 27
68010 25, 27
68020 27
8520 (see CIA)
AbleICR 93, 95
AddHandler 82
AddICRVector 92, 93
AddIntServer 52, 55
AIC 17, 18, 20, 21, 29, 31, 34, 37, 38, 52, 55, 57, 64, 65, 77
AIC bit number 19, 32, 37
AIC registers 17, 77
AIC support functions 80
AllocMem 50
AllocSignal 71
Amiga Interrupt Controller (see AIC)
assembly code 7, 44, 46
assembly language 46
audio channels 15, 82
audio example 84
auto-vector table 25
blitter 15, 41, 82
C 43, 44, 45, 67
Cause 52, 59, 99
chain termination 34
chaining 34, 65
CIA 14, 16, 21, 91
CIA example 97
CIA functions 93
CIA ICR 22, 91
CIA resources 22, 91
CIAICRB 96
CIAICRF 96
clear 77, 79, 83
ClearIntr 80
code pointer 49
compiling 67
Complex Interface Adaptors (see CIA)
COPER 58, 64
copper 15
copper example 73
CPU 20, 21, 23, 24, 25, 27, 31
CPU exceptions 26
CPU priorities 28
CPU priority level 27
CPU registers 27, 43, 45, 64, 84
CPU traps 26, 65
critical section 21, 81, 83
current priority level (CPL) 23
custom chips 14, 77
data pointer 49
data segment 45, 47, 64, 67, 84, 91
data structure 47
decoding interrupts 31
device drivers 12
device I/O 26
direct memory access (DMA) 15
disable 77, 79, 81
DisableIntr 80
disk 82, 92
Dr. Jekel 45
dynamic data segments 49
enable 53, 55, 61, 77, 79, 80, 81, 83, 91, 95
Enable_Abort flag 67
EnableIntr 80
example, audio 84
example, CIA 97
example, copper 73
example, serial port 87
example, signal and wait 71
example, simple 67
example, SWI 99
example, SWI message 100
exceptions 38, 40
Exec 8, 17, 20, 25, 26, 28, 31, 36, 47, 48, 64
Exec library functions 52, 71
Exec messages 100
Exec node 47
ExecBase 28
expansion bus 14
EXTER 22, 58, 64, 66
external hardware 34
external interrupts 16
floppy disk 92
floppy disk controller 15
flow of control 11
FreeIntr 51, 69
FreeSignal 71
global system state 26
handler 32, 52, 53, 82
handler environment 84
hardware priorities 20
ICR 91
ICR bit names 96
ICR vector 94
include files 48, 52, 55, 57, 77, 93, 96
initializing a handler 83
initializing an Interrupt structure 49
int_end 45
int_start 45
INTB 78
INTEN 80
INTENA 17, 18, 20, 31, 77, 78, 80
INTENAR 17, 61, 77
interrupt acknowledge 24
interrupt bit number 52, 55, 57
interrupt bits 78
interrupt code pointer 49
interrupt controller 17, (see AIC)
interrupt data pointer 49
Interrupt data structure 47
interrupt handler 53, 82
Interrupt Handlers 32
interrupt monitor 61

Interrupt Processing Level (IPL) 18
interrupt reentrant 26
Interrupt Server 33, 55, 58, 64
interrupt sharing 33
interrupt sources 15
interrupt structure 49, 54
interrupt vector 53
interrupt vector table 32, 54
INTF 78
INTREQ 17, 18, 20, 31, 36, 53, 55, 77, 78
INTREQR 17, 77
IPL 18, 20, 24
IPL lines 21, 23
Kickstart 7, 49
latency 29
library 7, 12, 52, 56, 58, 65, 67, 71, 84
list node 47, 48
MakeIntr 50, 69
Manx, Aztec C 6, 45, 67
master enable 20, 80
MEMF_PUBLIC 51
memory allocation 50
message 100
Mr. Hyde 45
multitasking 4, 22, 26, 36, 91
nested interrupts 29
nesting 29
NMI 21, 24, 55, 58, 64
node 47, 48
node priority 48
non-maskable interrupt (see NMI)
NT_INTERRUPT 48, 59, 99
NT_SOFTINT 48, 59, 99
OpenLibrary 93
OpenResource 92
Paula custom chip 14, 17
PORTS 22, 64, 66
priorities 36
priority group 19, 31
priority level 18, 22, 28, 30, 39, 59, 65
priority, node 48
priority, servers 66
program counter (PC) 24
programming language 43
programs 12
pseudo-priorities 20, 30
pseudo-priority 32, 36
RBF 82
reentrant 44
RemHandler 83
RemICRVector 93, 94
RemIntServer 52, 57
RemIntServer bug 58
request 77, 79
RequestIntr 80
resource 22, 91
ROM based interrupts 49
ROM Kernel Manual 3, 52
run-time environment 8, 43, 45
saving registers 27, 45
serial port 15
serial port example 87
server 33, 52, 55, 57, 64
server chaining 65
server chains 34
server environment 64
server handler 34, 53
server priorities 66
server registers 64
server termination 34, 66
SETCLR bit 78
SetFunction 58
SetICR 93, 95
SetIntVector 52, 82
SetSignal 71
sharing (see server)
Signal 71
Signal example 71
small memory models 45
software interrupts (see SWI)
sources of interrupts 15
spurious interrupt 31
SSP 27
stack checking 44
stack overflow 28
stack space 27, 32, 65
status register 24, 26
supervisor exception 26
supervisor mode 25, 26, 27, 38, 65
supervisor stack 24, 27, 28, 32, 38, 44, 65
supervisor stack bounds 28
supervisor stack frame 27
supervisor stack pointer 27
supervisor stack size 28
SWI 38, 39, 52, 59, 99
SWI example 99
SWI message example 100
SWI priorities 39, 48, 99
task 25, 26, 28, 30, 35, 38, 40, 70
task dispatching 36
task events 40
task exceptions 40
task scheduling 27, 36
task signals 44, 70
task stack 28
TBE 82
termination flag 56, 66
user stack pointer (USP) 27
Vector Base register 25
vector number 24
VERTB 15, 56, 58, 64, 66
Wait 26, 71
Wait example 71
zero flag 56, 65

An advanced programmer's guide devoted exclusively to the topic of interrupts on the Amiga – written by Carl Sassenrath, designer of the Amiga's multitasking architecture.

$14.95 U.S.